A Static Analysis Tool for Finding Buffer Overflows in C

University of Oulu
Faculty of Information Technology and Electrical Engineering / Degree Programme in Information Processing Science
Master's Thesis
Henri Hyyryläinen
24th May 2020

Abstract

This thesis presents a new static analysis tool for C and C++, that can detect some buffer overflow errors, which are dynamic memory use related errors that happen when a program tries to read or write past the end of a memory area. The tool is implemented as a plugin for the Clang compiler in order to leverage the excellent C and C++ parsing Clang has. The new tool is ran on Clang's abstract syntax tree (AST) representation, from which it is able to detect unsafe memory related operations that are in the analysed source code. A previous study by the author was done on static analysis theory and existing implementations in journal articles and scientific conference papers. One of the main findings was that there are no easily usable existing tools. For this reason This followup thesis set out to implement a new static analysis tool in order to start filling that found deficiency. The developed tool is available on Github at https://github.com/hhyyrylainen/smacpp.

Such a tool is important in software development as static analysis can reduce the number of bugs that slip through to released versions of software. If only manual testing and automated unit testing is used on software, it leaves many problems hidden that a static analysis tool could find. This is why static analysis tools are important as not using them increases the number of problems that software developers do not find. This thesis focuses especially on dynamic memory related errors as the type of problem that static analysis is used to find. This is because many memory related issues can be remotely exploited making it a very important aspect to get right. Memory unsafe languages are ones that do not guard the programmers against incorrect dynamic memory usage, C and C++ are widely used examples of these kind of programming languages. As these languages do not guard against memory errors, static analysis is a good addition to a development workflow to catch issues before they can be exploited.

The developed tool was tested with an existing test case set in order to verify that the tool can detect problems correctly in concrete programs. Because this test set contained only C programs, the developed tool focuses on them, instead of also handling C++ specific issues. In addition to the first test set another set was used for evaluating the performance of the new tool once it was completed. The new tool, when combined with Clang's analysis as the new tool was designed to compliment Clang's own analysis capabilities, is able to detect 4 more issues in the first test set, without adding any false positives. This means that the combination is useful. Unfortunately none of the tested tools were able to pass any test cases contained in the second test suite. The new tool also increased the number of false positives when combined with Clang, but this is likely due to many of the test cases missing the expected entry point, "main". In addition to the new tool, this thesis presents the way it was designed and how it uses Clang's libraries to aid in the development of a static analysis tool.

Keywords
static analysis, program analysis, memory error, type systems, static analysis tool

Supervisor
Ph.D., University Lecturer Antti Siirtola

Foreword

I’m glad to have finally finished this thesis, as it was a struggle. I started on this topic a year ago, in the form of my bachelor’s thesis. At the time, I was very excited about static analysis, but now that excitement has faded. The hardest part about doing this thesis has been keeping up the motivation needed to work on this. It’s very hard to stay motivated about a long-term project. Especially scientific writing is very demotivating. Perhaps I could have made doing a thesis easier on myself, had I selected a different topic. Recently, I took some courses which had, almost, ready-made thesis topics that seemed very good and had many different parts to them, that may help with keeping up interest. So I guess as advice to future thesis writers, I’ll say that you should really think about your topic and select one that’ll be interesting to you even a year after starting.

That’s enough random thoughts about the writing process, now onto the thank yous. First off, I would like to thank my supervisor, Antti Siirtola, for his comments on the draft versions of this thesis, as well as for finishing supervising this thesis, despite the changed circumstances. I’d also like to thank everyone else who offered me feedback on this thesis.

Henri Hyyryläinen

Oulu, April 14, 2020

Abbreviations

Static analysis is an approach to finding defects in software by analysing the static structure of the program, for example source code. Usually done with a static analysis tool that goes through the source code and reports problems.

Dynamic analysis looks for problems in a program while it is running by adding checks that ensure that dangerous operations do not happen.

Buffer overflow is an error where a program writes past the end of a memory area. It is caused by incorrect programming leading the program to not restrict itself to within the bounds of a memory area.

Abstract syntax tree (AST) is an internal representation of source code that a compiler builds in order to facilitate various operations, like optimization and code generation, to be ran on it. The AST can also be provided to external tools from the compiler to facilitate the tools understanding source code better than if they had to reimplement source code parsing.

Recall / sensitivity is the fraction of true positive reports of the total number of positive reports: $\frac{TP}{P}$ (Moerman, 2018, p. 7). 100% is the perfect score.

Precision is the fraction of the true positives of the total detected errors (true positives and false positives): $\frac{TP}{TP+FP}$ (Moerman, 2018, p. 7). The higher the percentage, the better.

SMACPP is the new static analysis tool developed in this thesis. Available on Github: https://github.com/hhyyrylainen/smacpp.

Contents

1. Introduction

The modern world runs on software. As such is it important that software is reliable (Hiser, Coleman, Co, & Davidson, 2009). Reliable software is able to perform its intended function under certain conditions with a high enough success rate (Zhogolev, as cited in Frolov, 2004.). With such a definition, software does not have to be completely error free, just good enough to meet the required reliability. Also a very important aspect of software is security (Frolov, 2004). Software security is about preventing compromises to the integrity of software (Hyyryläinen, 2019). For these, and other, reasons it is important that attention is paid to how software is developed. Hiser et al. (2009) says that current software development practices are lacking and many kinds of memory related errors are made. These errors reduce software reliability and security. Better tooling can help with these problems.

Static analysis tools are a category of tools that help in software development by detecting errors from a program's source code or some other representation. Compared with static analysis, dynamic analysis is performed by running the software and seeing if it does something incorrect. Static analysis is done without running the program. As such it can explore the entire possible space of possible program execution paths and detect errors that maybe very hard to find by tweaking the program inputs to get it to execute many different paths through the code. When these tools are incorporated into a development workflow, they reduce the amount of errors in the finished software, making it more reliable (Dhurjati, Kowshik, Adve, & Lattner, 2005). Black (2012) even argues that any ethical software development must use static analysis tools, otherwise the developers are being irresponsible by not ensuring software quality.

Static analysis is especially helpful in memory unsafe languages where a simple programmer mistake can have catastrophic consequences, for example compromising the security of an entire server running many different services. Memory unsafe languages are used for their speed, for that reason hybrid or dynamic analysis tools are not optimal as they incur runtime performance costs, sometimes even very heavy. This is why incorporating static analysis tools to C and C++ software development workflows is important part of making good quality software. The found journal articles and conference papers did not provide their developed tools in a usable form. That is why this thesis presents a new open source tool that, while quite basic, could serve as a basis for a more advanced tool. The new tool could even be used as-is to detect some issues.

This thesis is based on my bachelor's thesis (Hyyryläinen, 2019) and builds on top of it by creating a new static analysis tool that is open source and is available online in order to make it as easy to access as possible for future researchers and also any interested software developers. This new tool was developed keeping in mind the findings of the previous study. In order to keep the amount of unnecessary work down the tool uses the clang compiler to parse source code. The relevant portions of the literature review done in it are included in the prior research chapter of this thesis.

The research method used in this thesis is design science. The research method is described

in detail in Chapter 4. The main research question is:

> How can we create a static analysis tool that complements Clang's static analyzer in detecting buffer overflows?

All of the used test suites are pre-existing ones. The first of these test suites, Moerman's (2018) test suite, was used to direct the development of the new tool in a test driven development fashion.

In the next chapter a detailed description of static analysis is presented, as well as a comparison against dynamic analysis, which is a related technique, is made. Then in the next chapter the literature review continues, first the existing literature from journals and scientific conferences is discussed. And after that, in the second part of that chapter, other found material such as Moerman's (2018) thesis, which is not peer reviewed material, is discussed. In the methods chapter it is explained how design science research is followed in this thesis. Then the rest of the chapters are dedicated to introducing the new tool, the design decisions, functionality, and features. After introducing the tool, the next chapter compares how well the new tool performs against other open source tools in the area of detecting buffer overflows, the results of this analysis are also discussed. Finally, the implications of this thesis are discussed, as well as the limitations, and a direction for future research is presented. The major findings are also recapped in the last chapter.

2. Static Analysis

This chapter introduces the concept of static analysis as well as its properties based on the found literature. Then static analysis is compared with dynamic analysis in terms of their strengths and weaknesses, it is also discussed how these approaches have been combined in the earlier literature. This chapter contains large parts of text originally published in Hyyryläinen (2019).

2.1 Description of static analysis

Bugs happen when developing software and in order to prepare for this inevitability, it is a good idea to employ static analysers (Black, 2012). It has long been believed that detecting errors in software before it is ran is beneficial (Das, 2006; Sokolov, 2007). Especially in safety-critical systems where detecting problems before runtime is absolutely vital (Dhurjati et al., 2005; Kowshik, Dhurjati, & Adve, 2002). Traditionally testing was the method for detecting errors in software but new tools such as static analysers and dynamic protection have been developed (Frolov, 2004). Static analysis tools work by attempting to build a flow graph of all possible executions of a program (Grech, Fourtounis, Francalanza, & Smaragdakis, 2018; Sokolov, 2007). And from this the analysis tool attempts to detect problems. Dynamic analysis tools, in contrast, insert runtime checks into a program to catch problems.

Static analysis tools are programming error detection tools that are ran on the static structure of a program, hence the name. They can detect many kinds of bugs like memory leaks and out of bounds memory access (Ciriello, Carrozza, & Rosati, 2013). This thesis focuses especially on memory bounds checking. This kind of analysis is performed before program execution (Frolov, 2004). It can be done on different representations of the program, either on the source code or a compiled form. Analysing the source code is the most often used form as it is the most informative representation of a program. (Frolov, 2004.). However many of the papers referred to in this thesis have gone the route of running on machine code or byte code. One of the reasons for this is that there are situations when memory errors need to be detected in software without access to the source code (Hiser et al., 2009). For these cases, the memory error detection system introduced by Hiser et al. (2009) works on binary executables. Running the analysis on compiled programs also has the advantage that the tools can work on many compiled languages much easier than tools that work on source code, as the source code of different programming languages can differ greatly, whereas the generated machine code uses is built from the same instructions.

Static analysers are used for scanning through large amounts of code for problems that would be nearly impossible to find manually (Ciriello et al., 2013). Memory leaks are one such example, which does not always affect program functionality but makes the software use much more memory than it actually needs (Ciriello et al., 2013; Sun et al., 2018; Xu & Zhang, 2008). Of course it is still possible for memory leaks to cause software to fail

that is long running and memory-intensive (Heine & Lam, 2003; Xie & Aiken, 2005; Xu & Zhang, 2008). This might also lead to the performance degradation of an entire system, and for example failure to launch new processes as well as excessive swapping slowing down everything running on that system (Sun et al., 2018). Though, once you know that there is a memory leak, it is possible to use analysis tools like Valgrind to point the programmer to the spot where the non freed memory was allocated.

Static analysis is a valuable tool, as through automating many manual inspections, it allows a small team to cope with large programs without excessive costs (Wendel & Kleir, 1977). With traditional code review the accuracy and number of problems detected is good but the desire to reserve humans for more difficult tasks lead to the creation of static analysis tools. Good tools can quickly identity weak spots and possible bugs in code, this is something that is not possible with humans checking the code. Basic tools can detect some issues like floating point comparisons, more advanced tools are needed to detect issues with memory. (Sokolov, 2007.)

Static analysis is also important in keeping bugs away when software is being modified. Especially with legacy software, because even if it has tests, it still might contain many undetected defects (Ciriello et al., 2013). That can then surface when the software is modified (Ciriello et al., 2013). Another reason is that programmers can make mistakes when doing maintenance or upgrades to existing software (Landwehr, Bull, McDermott, & Choi, 1994, p. 223; Sokolov, 2007). When software is being modified it should be reviewed as carefully as when it was originally written (Landwehr et al., 1994, p. 223), but often this is skimped on. The situation gets even worse if the system that is being modified has no regression tests (Sokolov, 2007). This is another good spot to use static analysis to fill gaps in tests. Tools are a major help here as they can be used to automatically and repeatedly check millions of lines of code (Black, 2012). Though, the recommended solution to working with legacy software is to first write tests for the legacy system, in order to be able to verify that the changed system works the same (Sokolov, 2007).

Static analysis is especially beneficial when the used programming language has a type system with which the compiler can ensure program correctness. A limitation in static analysis tools is the limited information their defect reports give, as they do not describe the scenario needed to trigger the problem. This leads to programmers not fixing some number of the problem reports. (Das, 2006.). Also to be practical static analysis must fit seamlessly in the software development workflow (Ciriello et al., 2013).

2.2 Comparison between dynamic analysis and static analysis

Both dynamic and static analysis, approaches have upsides and downsides (Frolov, 2004; Weber, Shah, & Ren, 2001). Static analysers can detect many problems in software like performance bottlenecks, safety violations and security vulnerabilities (Bodden, 2018). As discussed before this detection is made on the static structure of the program before running it. Whereas dynamic analysis, also called dynamic protection, inserts guards into the program, that check, for example, the validity of pointers during runtime. So the detection of problems in it happens when the program is ran. This is the main difference between them, but many approaches combine both into the same tool, more on that in Section 3.3.

The main advantage of static analysis over dynamic analysis is that it is complete. The

completeness here means that the generated flow graph contains all possible executions of the target program. This is very effective but suffers from two major downsides: it is expensive to compute for substantial software, and it lacks precision. Precision is the ratio of incorrectly detected issues to the number of correctly detected issues. The lack of precision in static analysis is caused by the predicted flows not entirely matching actual flows when running the program. (Grech et al., 2018.) Whereas in dynamic analysis the observed program behaviour depends on the user input it is given (Sokolov, 2007) and thus is limited by how comprehensively the software is tested with different inputs.

The major downside of static analysis is the false positives, among valid reports, it usually generates. This causes wasted time spent investigating these false positives. Often when increasing the amount of actual problems static analysis tool finds, it also finds more false positives, which means that its usefulness does not necessarily increase. (Ciriello et al., 2013.) With many false positives developers might not feel like using static analysis is worth their time (Hovemeyer, Spacco, & Pugh, 2005). However many existing static analysis systems are lacking in branch awareness and thus their detection accuracy suffers and they might, for example, not be able to find memory leaks (Sun et al., 2018). Not filtering out infeasible paths through the program is a common source of this inaccuracy (Hovemeyer et al., 2005). This is a place for a lot of potential improvement. But Ciriello et al. (2013) goes as far as to claim that a perfect static analysis tool can never exist and all problems can not be found due to the nature of computing.

Dynamic analysis, in the form of automated or manual testing done by running the program, can detect some aspects of the program much easier than static analysis, as with static analysis there are scalability problems when trying to infer all possible execution flows (Grech et al., 2018). Grech et al. (2018) propose a hybrid approach to static analysis where the static information is augmented with dynamic runtime information in order to reduce false positives and increase the performance of the analysis. They thus conclude that in program analysis there are three competing quality properties: completeness, precision, and scalability. Their approach is designed to focus on precision and scalability at the cost of completeness as the dynamic information they add to the static analysis process cannot capture all possible program execution paths. Compared with earlier tools that they mention, their approach replaces parts of the static analysis with dynamic facts in order to improve the scalability. This dynamic information is derived from heap snapshots. A huge limitation in their work, in applicability to the topic of this thesis, is that their tools only work with the Java Virtual Machine (JVM). However their approach to not modeling the heap fully in static analysis and instead only relying on information from actual heap snapshots, without modeling writing operations, may be applicable to C and C++ as a generalized concept.

The major disadvantage of a dynamic analysis system is the degraded performance during runtime. Another disadvantage is that most systems cannot correct the program behaviour and for example can only terminate the program to prevent more harm from being done. The advantage is the simplicity of application. Combining this approach with static analysis allows reducing the number of places that need protection. In this approach, static analysis systems classify parts of a program in three categories. The first is unsafe fragments which are guaranteed to contain errors. The second is safe fragments which the static analysis could prove to be safe. The final category is the potentially unsafe fragments which could not be unambiguously classified into the other categories. This class needs runtime protection to alleviate the potential errors in them. A better static analysis might be able to reduce the number of these program fragments. Usually there is such

a large amount of the potentially unsafe fragments that manually checking them is not feasible and this is where dynamic protection is a good tool for filling the gap. This way the dynamic checking is much cheaper and there is a reduced number of false positives, compared with plain static analysis. (Frolov, 2004.) However, the dynamic checks do still reduce the runtime performance of the software, even if the cost is partly mitigated.

3. Prior Research

In this chapter, the found prior research is explored in terms of their results and findings, as well as used as the basis for motivating the need for the artifact developed in this thesis. This chapter also contains large parts of text originally published in Hyyryläinen (2019). In the next section, the impact of memory issues on software security is discussed and how static analysis can help. Then the results of the previous studies are discussed as well as any existing static analysis tools that the studies presented. Finally other prior work, that was not published as scientific articles or conference papers, is discussed.

3.1 Impact of memory issues on security

The aim of software security is that software continues to function correctly even while under attack (McGraw, 2004). Memory errors do not always expose a way for an attacker to exploit them but even in these cases an attacker could potentially exploit these problems in order to perform a very effective denial of service attack (Sun et al., 2018; Yong & Horwitz, 2003). However, when memory errors in programs exposed to the internet are exploitable, they can lead to the attackers gaining control over an entire system (Oiwa, 2009). So from a safety viewpoint it is also very important to make sure these types of errors are found and fixed. Or mitigated some other way, for example with dynamic checks around memory access. This means that not using static analysis may make software much more vulnerable to attacks, than if a static analysis tool was used that can detect various issues.

C and languages similar to it employ manual memory management (Xu & Zhang, 2008). These types of languages are not memory safe languages as they require the programmer to explicitly request and free memory when needed instead of the language taking care of it automatically. C and C++ are examples of these kind of, memory unsafe languages (Kroes, Koning, van der Kouwe, Bos, & Giuffrida, 2018). Most common flaws in programs are memory related (Hiser et al., 2009). This does not apply to languages designed to be memory safe as their language design is made to counteract these issues. The most common issues, in not memory safe languages, are array bounds checking problems (buffer overflow and underflow) (Chess, 2002; Hiser et al., 2009; Kroes et al., 2018; Oiwa, 2009), not releasing memory (memory leak) (Heine & Lam, 2003; Sun et al., 2018; Xie & Aiken, 2005; Xu & Zhang, 2008), releasing memory too soon (also called dangling pointers) (Heine & Lam, 2003; Hiser et al., 2009; Oiwa, 2009), and problems with using uninitialized data (Hiser et al., 2009). This thesis focuses on buffer overflows as the type of issue that the new tool is made to detect, in order to keep the scope of this thesis manageable.

Even though a lot of research has gone into fixing these types of issues they are still prevalent. Buffer overflows are particularly troublesome as they are quite often easy to exploit and use as a point of entry for a worm. (Kroes et al., 2018.) Memory leaks are another kind of issue that is still widely problematic in many widely-used programs (Xie & Aiken,

2005). A common attack is to exploit a buffer overflow and replace the return address to be in data provided by an attacker. This then causes the program to continue executing the attacker's code. There are ways to protect programs against this type of attack, some of them have high runtime costs. (Yong and Horwitz, 2003.) In fact most attacks are not done by finding novel ways to compromise systems rather most attacks are repeated exploits of already known problems (Evans & Larochelle, 2002). Out of the 190 security attacks referred to by Evans and Larochelle (2002) only 4 are problems in cryptography, most of the rest are problems like buffer overflows and string format errors. In another study referenced by Ganapathy, Jha, Chandler, Melski, and Vitek (2003) buffer overruns where the top vulnerability in UNIX systems.

Static analysis and dynamic protection tools help in creating software that is resistant to attacks (Weber et al., 2001), as a single mistake by a careless programmer, is able to undermine the security of the entire system (Ganapathy et al., 2003). These mistakes can be hard to catch without any help from tools. In addition to tools, secure software development can also be helped by an execution environment, for example the Java virtual machine. This is not without problems either as the Java virtual machine can also contain security vulnerabilities. (Pomorova and Ivanchyshyn, 2013.)

3.2 Results from the literature

This section goes over the major findings about static analysis from the literature review in Hyyryläinen (2019) in light of the problems identified in the previous section and chapter. The next section focuses on the existing, concrete tools from the previous work.

The need for reliable software has resulted in a lot of research being done to identify ways to detect the violation of memory safety properties in programs. This property can be generalized as requiring that each value created by event A must reach exactly one event B, in every program execution flow. This type of property is called source-sink property. With memory errors, the event A is the allocation of memory and event B is the respective deallocation of memory. In a program that fulfils this property, each memory allocation is freed exactly once. (Cherem, Princehouse, and Rugina, 2007.) This does not encompass every type of memory error, this is only for memory leaks and double frees, but extending this definition with additional access events that must happen between events A and B this model is also valid for other types of memory errors like use after free and the use of unallocated memory. Programs that fulfil this property can be considered less likely to contain a memory related problem.

Weber et al. (2001) say that static analysis techniques, that existed at the time they wrote their paper, often worked on a really abstract level. For example by only just saying that a buffer overflow is possible, instead of being able to derive the program inputs that would result in that behaviour. Their technique also suffers from this limitation. But they claim that this is still useful information in the space of program security because, if it is possible to theoretically for the program to do a buffer overflow, it is good to fix it. Creating an accurate static analysis is not easy and in fact it is very easy for static analysis tools to miss problems when the analysed program contains complex branches (Sun et al., 2018).

Frolov (2004) uses a hybrid approach of combining static and dynamic analysis. In their approach, they use static analysis to prove some operations safe and use dynamic checks to protect other operations similar to Oiwa (2009). In addition they compare the advan-

tages and drawbacks of static and dynamic analysis. With static analysis, the developers must address the issues, unlike with dynamic analysis that can insert automatic checks for preventing the errors, and this may be difficult as many issues found by static analysis cannot be automatically decided. The program execution can depend on many environmental factors that the static analysis simply cannot predict. Thus it is necessary for the static analysis to cut corners here and not be entirely accurate in terms of all possible program executions. This can be mitigated with three approaches: asking the developer questions, finding a superset that also contains many false positives, and referencing a database with information regarding the runtime environment. All of these are labour intensive. Even the superset finding, as that results in a lot of false positives, that must be manually checked. The most promising of these is creating and maintaining the knowledge base about the program environment. In the knowledge base everything that the static analysis cannot infer, needs to be provided. For example, the results of system calls or network requests are the kind of information that needs to be added to the knowledge base. (Frolov, 2004.)

The final important finding of Frolov (2004) was that the hybrid approach they described can be iterated on. Once the approach has been used once the performance and reliability of the system can be further improved. This can be done by manually marking parts of the program safe in order to reduce dynamic checks. This helps the process get started as it can be very difficult to create a static analysis algorithm that can decide enough code fragments to be safe in order to not catastrophically affect the performance. Then it is possible to iteratively design the static analysis algorithm to classify more and more fragments unambiguously. In their example, they present the case of protecting against output format string vulnerabilities. First the analysis can be made to just check static variables in the current translation unit for the used format string. This will miss some safe uses, but will already limit the number of potentially unsafe fragments. To further improve this the analysis can be extended to include information from other translation units in order to be able to determine the format strings of more output function calls. (Frolov, 2004.)

Frolov (2004) also includes an experimental study that shows that even with just 4 steps and a code base of 500 000 lines of code the number of false warnings were reduced from 2618 to 35. In this experiment they only focused on format strings. Out of all the warnings only 0,15% were actual problems. At first they just counted all code fragments with output functions as unsafe, this resulted in 2618 potentially unsafe fragments. Then they implemented a basic analysis for detecting the use of constant format string in a single translation unit. This reduced the potentially unsafe fragments to 1150. Then they added detection of constant strings from other translation units and detection of print wrappers bringing the number of potentially unsafe fragments down to 35 that could then be manually examined in a day. They used a dynamic part to the analysis that protected the program from the start. It started at having a 7% impact on performance and 0% at the end when all of the calls through the protection layer were removed.

Static analysis can be used, in addition to catching memory errors, to verify that best practices and guidelines are followed in C++ (Sokolov, 2007). Sokolov (2007) also brings up the company culture aspects of static analysis tools. Mainly that they must be incorporated into the workflow in order to get used properly. It is no use having good tools if people only sporadically use them. Ciriello et al. (2013) presents similar models to Sokolov (2007). In their work, they explore the effects of applying the principle of continuous integration to static analysis. They call their approach continuous code static analysis. They describe it as running the static analysis, which can take multiple hours, in a similar fashion as

continuous integration on a build server after code has been committed. In order for this to be a viable approach there needs to be high quality static analysis tools available for developers to use.

In the case studied by Ciriello et al. (2013) they made the C++test tool run during the weekends and provided reports for the developers to fix for the next week's release, the company used a weekly release schedule. They also identified the need to keep false positives as low as possible. They also present that it is possible to track the direction the software is headed quality wise with continuous static analysis. This is done by tracking the number of issues detected from each software version. Then the direction can be determined from the last few points. If the trend is towards more issues then corrective measures can be implemented.

Bodden (2018) proposes that future static analysis tools could be developed to be self adaptive in order to improve their scalability. Scalability of current static analysis tools is a problem as the size of software grows. Current tools are written in general purpose programming languages with a limited set of customizability they can select based on the analysed program. To solve this problem Bodden (2018) proposes that static analyses should be created in a dedicated intermediate representation, for example as a graph problem. This provides much better chances for optimization than general programming languages. Additionally the analysis process should continuously adjust itself similarly to how just-in-time compilers work. The result would be that for each static analysis problem the static analysis tool would generate a highly optimized analysis. (Bodden, 2018.)

Bodden (2018) does not present a working implementation of their ideas. They only present the core idea in the hopes that the research community can help in the implementation of their ideas. They present the core concepts and the challenges they anticipate potential implementors will run into. At the core of the suggested algorithm is a new declarative definition language crafted specifically for creating static analysis tools. The design of this language needs to be balanced with regards to be able to express a variety of static analysis techniques but at the same time being restricted enough to offer potential for optimization. Then in their design this representation is compiled into a high level intermediate implementation that can be optimized. Then there would be also a low level representation that would be optimized in a way that takes the target program into account. Then this representation is ran on the target program and profiling information is collected that can then be used to tweak the analysis to be more efficient, with help from a human. Then the final part of their design is the low level just-in-time compiler part that would constantly monitor the running analysis for bottlenecks and change to algorithms that are more efficient in the current situation. (Bodden, 2018.)

Lee, Hong, and Oh (2018) present a static analysis technique for detecting memory deallocation errors in C programs. In addition to detecting the issues their approach can automatically generate fixed code. This is based on solving an exact cover problem derived from their static analysis. The solution is a set of free statements that deallocate each piece of allocated memory exactly once. This makes it possible to make a fix that does not introduce new errors. This makes fixing these issues much easier as when manually fixing the programmer must consider all possible paths in order to not introduce a new problem, for example a double free, while trying to fix another problem.

Weber et al. (2001) is another paper where the authors developed a new tool that seems to have no source code available. They focused on developing a static analysis tool for

detecting buffer overflows. They based their work on earlier analysis techniques that had serious limitations in terms of them not taking program flow or scoping into account. Weber et al. (2001) say that by taking program flow into account it reduces the number of false positives significantly. Their algorithm is based on determining the maximum lengths of data written to buffers and then determining if the buffer is big enough in all possible cases. A major limitation in their tool is that it must be given a list of potentially dangerous program statements to analyse, so it cannot be used on its own.

In general regarding static analysis, Weber et al. (2001) present that as program complexity increases the number of possible control flow paths increases dramatically. This is the reason why large programs cannot be accurately analysed with static analysis tools using a graph presentation, like used by Weber et al. (2001). In their experimental results, they show that their developed tool can reduce the number of hits from a tool that scans for potentially unsafe function calls by 25% to 60%. They only tested on three programs, resulting in the large variance of the results.

In contrast to the other papers discussed Hovemeyer et al. (2005) present a simpler analysis that tries to just find null pointer problems. They show that their tool is effective at finding actual problems without a huge number of false positives, despite the analysis being simple. Their tool is for analysing Java bytecode and it is still available. Their analysis focuses on reducing the amount of false positives by conservatively assuming that unknown values are not errors. They also greatly limit the size of their program flow graph by leaving out complex interaction between functions and determining actually executed polymorphic functions. Instead their tool assumes that many variables are unknown and any concrete method in the case of polymorphic methods could be executed, but once again they limit the amount of warnings they give as their tool does not have enough information to give good warnings. They do mention that their tool will generate warnings from polymorphic code where all the possible implementation methods can cause a problem.

3.3 Existing tools for error detection

This section explores the static analysis tools that were developed in the existing literature. Other tools that were referenced in the previous works are also briefly mentioned.

Many of the articles referenced in this thesis present static analysis or hybrid static and dynamic analysis tools that the authors developed. Sadly, it seems that none of these tools, that work on C++ code or x86 machine code, are available *and* open source. This is the most major limitation in the existing literature. Still, a few tools were found, but they are unmaintained and bordering on obsoleteness, or they were for a memory safe language. See the section about Moerman's (2018) thesis for a review of a few open source static analysis tools.

It seems that many of the papers set out to only create a proof of concept implementation of their algorithms and not to create a useful tool ensuring that others could benefit easily from their work. As such if anyone wants to use the findings of these papers to improve their software, they would have to reimplement the tools from scratch. For this reason the tool developed in this thesis is made open source so that it can be built on top of.

However, Lee et al. (2018) reference an earlier tool that has source code available, but the improved tool they themselves created is not available in source code form. Though,

there are existing tools, they are proprietary. An example of this is that Ciriello et al. (2013) present an approach where a static analysis tool is automatically ran on committed code on a server. Then the generated reports from the long tool runs, over the weekend, are delivered to the developers. This is an example of how the tool can be incorporated into a workflow. Unfortunately, the tool, C++test by Parasoft, they used is proprietary. Ciriello et al. (2013) describe the tool as aiding developers prevent software defects by utilising rules that are tuned to find code patterns that lead to problems. As a proprietary tool, it's impossible to say what approaches are used by it and thus it is not possible to use their results in building a base for future research. In their tests, 14% of reported issues were false positives.

The tool presented by Lee et al. (2018) is available, but only in binary form with no source code easily findable. Lee et al. (2018) mentions LeakFix tool that has a similar goal as their tool and that has source code available. It is even quite recent with the source code dating only back to 2015. However, the setup process requiring building their modified source version of an outdated compiler makes this less useful.

The compiler for memory safe C developed by Oiwa (2009) is available but the last update to the project has been in 2010 and it thus depends on outdated outside libraries. I was unable to build this program due to the Bohm garbage collector not being detected by the configure script, likely due to the version I downloaded being too new. It is probably possible to update the code to work again, but it may take some considerate amount of work. Alternative outdated versions of the required libraries might allow building the software. Though even then the compiler is outdated as multiple new C standards have been published since its last update.

Sun et al. (2018) say that in their experience existing static analysis tools, that collect information from scanning source code, are lacking in accuracy and efficiency. Kroes et al. (2018) agree that overheads caused by existing tools are unacceptable. They also say that the existing tools suffer from poor compatibility. This is a huge problem, as mentioned earlier, that low accuracy results in a lot of false positives and wastes time of developers.

Grech et al. (2018) mention HeapDL and Tamiflex tools, in addition to the tool developed in their paper, that implement some static checking for Java programs. These tools were not researched further as the focus of this thesis is to explore the feasibility of static analysis mainly for memory unsafe languages.

3.4 Other prior work

In the previous sections only journal articles and scientific conference papers were discussed. In this section, other prior material that is not as high quality in terms of peer reviews is discussed. This material was not included in Hyyryläinen (2019) as the literature search done for that work was limited to articles in Scopus' database.

Moerman (2018) studied how well various open source static analysis tools performed on some test programs consisting of quite little source code. The tools they compared where Clang static analyzer, Infer, Cppcheck, Split, and Frama-C. They found that the best tools were Clang and Infer, with Frama-C also having quite good issue detection but worse usability overall than Infer. Many of the problem categories could be correctly detect by only a couple of the tools, or just one (Moerman, 2018, p. 21). This means that the tools

have areas that they are good in and others where they are bad at. In addition if better coverage is wanted, many of the tools would need to be used at the same time. It was also found that the precision and sensitivity of a tool varies in different test cases, meaning that the tools are not equally good at finding issues in different circumstances (Moerman, 2018, p. 25). Especially difficult areas for the tools to detect are places where loops contain writes, there are recursive functions, or global variables are used (Moerman, 2018, p. 25).

The test suite by Moerman (2018) can be used to test how well a static analysis tool can find problems in a program where all the function arguments can be determined from the provided code (Moerman, 2018, p. 16) That test suite is also used in this thesis in order to develop a new static analysis tool and prove that it can find issues in source code. The test suite contains multiple versions of the same code with very small changes between a version that has a problem and a correctly behaving version. The test is considered passed if a tool gives an error for the incorrect version but not for the correct version. The test suite can also be used without entry points to test if a tool is able to detect that there exists a set of dangerous inputs for a function. (Moerman, 2018, pp. 11–14, 16.) This makes it very easy to make a script for running the test suite with different tools, only some logic is needed to detect that the tool does not issue errors or warnings for the correct version but *does* issue warnings for the incorrect version. The existing test suite used by Moerman (2018) is no longer available as the URL provided to download it is no longer valid. Thus it cannot be used for further testing any tools. The test cases including loops in the Moerman's (2018) test suite are more difficult for static analysis tools to correctly handle (Moerman, 2018, p. 25). So if a tool is able to cope with the harder cases it should also be able to handle the easier cases.

A point that was mostly overlooked in the other references is that the time it takes for a tool to run is an important characteristic. If it is very slow to run a tool locally on a developer's machine, the feedback cycle will be much slower, which may hinder the usage of static analysis (Moerman, 2018, p. 4). This also means that the static analysis tools can make a trade-off between running time and analysis precision (Moerman, 2018, p. 4). For analysing Gimp 2.8.22 source code, it took Clang only 22 minutes to analyse, for Infer it took 51 minutes and for Cppcheck it took 106 minutes (Moerman, 2018, p. 31). The latter of these times start to be very developer flow breaking, and cannot be well incorporated as part of normal work on the software, instead the tool will need to be run, for example, by a continuous integration server.

Moerman (2018) tested how well the tools performed by tracking the precision and recall of the tools in addition to checking how many issues each tool reported. Additionally they measured the performance of the tools and also the user experience of the tools, properties like how good the error messages are and what information needs to be passed to the tool. As part of the user experience, how well the tool can be integrated into a development workflow was also analysed.

Out of the tools tested by Moerman (2018) 4 have had a recent release. Splint has had not had a new release since 2007 (Moerman, 2018, p. 9). So similarly to the many other tools discussed in the previous section is not usable any more. Splint also failed to parse many included system headers making it impossible to analyse some programs (Moerman, 2018, p. 19). Because of that and low performance Moerman (2018) skipped running many tests with Splint. This basically concludes that Splint is unusable for any real world analysis needs. In the test by Moerman (2018) Clang had issues with detecting problems inside loops, in addition to not being able to detect buffer overflows at all. These are the most

promising places where an addition to Clang could be done to improve its usefulness. Clang analyzer and Frama-C were selected as the tools for comparing the new tool against because they had the best issue detection rate in Moerman's (2018) tests. Both were also quite easy to get running however I did encounter some usability issues with Frama-C.

4. Methods

Design science is the research method used in this thesis. How that method was used in this thesis is covered in this chapter, but first the research question is explained, then a description of how design science research works is given and then how it was customized for this thesis is explained.

The main research question is: how can a static analysis tool be created that complements Clang's static analyzer in detecting buffer overflows? In order to answer this question a new tool was created and evaluated. Design science research was selected because the goal of design science research is to produce new, innovative artifacts as its research output (Iivari, 2015), that that extend the boundaries of human and organizational capabilities (Hevner, March, Park, & Ram, 2004). This means that design science lines up nicely with the goal of making a new static analysis tool as an artifact. Hevner et al. (2004) focuses on talking about research related to information systems, but their description is extendable to other types of software as well that design science is applicable to, such as the software developed in this thesis.

Design science is related to behavioral-science and together they are important in furthering what is possible with information systems in terms of increasing the effectiveness and efficiency of an organization. This is due to information systems being used in organizational contexts, where many human factors need to be taken into account. Behavioral-science tries to explain these human behaviors, which affect how well an information system can be implemented. Behavioral-science has its roots in natural science research methods, whereas the design science paradigm has its roots in engineering. Design science research seeks to create innovations that, among other things, define ideas, practices, and products that aid in the effective use of information systems. While these artifacts are artificial they are not exempt from natural laws or behavioral theories, meaning the artifacts' creation relies on existing theories. (Hevner et al., 2004.)

In design science research, what is meant by artifacts contains many subcategories. These are constructs, models, methods, and instantiations. Constructs consist of vocabulary and symbols. Models are abstractions and representations. Methods are algorithms and practices. Finally, instantiations are implemented systems as well as prototypes. These terms are used to enable IT researchers and practitioners to understand the problems in developing information systems as well as to address problems with them. (Hevner et al., 2004.) The artifact created in this thesis is an instantiation as it is a concrete piece of software.

Design science is followed in this research by following the seven guidelines provided by Hevner et al. (2004). The guidelines are listed in Table 1. They are explained in the next paragraphs. After descriptions of the guidelines how the guidelines are followed in this research is explained.

Table 1. Design science research guidelines by Hevner et al. (2004)

Guideline 1	Design as an artifact
Guideline 2	Problem relevance
Guideline 3	Design evaluation
Guideline 4	Research contributions
Guideline 5	Research rigour
Guideline 6	Design as a search process
Guideline 7	Communication of research

Guideline 1, design as an artifact, says that the result of design science research is a purposeful IT artifact, that is created to address an organizational problem. In order for it to be useful, it must be described effectively so that it can be implemented in an appropriate way. Here artifact must be one of: a construct, a model, a method, or an instantiation. Most of the time artifacts are not full-grown information systems used in practice. Instead they are usually innovations that enable the creation of concrete systems to be used in practice. (Hevner et al., 2004.)

The second guideline, problem relevance, means that design science research must work towards the goal of acquiring knowledge that enables the implementation of software solutions to unsolved and important business problems (Hevner et al., 2004.). The results include a combination of technical, organizational, and people-based artifacts (Hevner et al., 2004), not just directly the goal-fulfilling results, but also intermediate results.

The third guideline, design evaluation, says that the designed artifact needs to be rigorously demonstrated with well-executed evaluation methods. This is to show the utility, quality, and efficacy of the artifact. The artifacts should be evaluated in a real business environment where the environment places constraints on the artifact and integrating it in the business environment is one evaluation criterion. In the evaluation of IT artifacts, the following criteria can be used: functionality, completeness, consistency, accuracy, performance, reliability, usability, fit with the organization, and other relevant quality attributes. When appropriate the artifacts can be evaluated mathematically. In an iterative process the evaluation step provides information back to the construction phase. Once the produced artifact is evaluated as good enough the development process ends. (Hevner et al., 2004.)

Guideline 4, research contributions, is about making sure that the performed research work contributes back to the knowledge base. There are three ways that a design science research contributes back. The first is through the designed artifact, which is the most common way. The artifact itself, when it enables new solutions, contributes knowledge. The second way is by creating foundations in the form of constructs, models, methods, or instantiations, that extend the existing foundations. The third way is through the development and evaluation of methodologies, by for example developing new evaluation metrics for design science to use. (Hevner et al., 2004.)

The fifth guideline, research rigour, is about applying rigorous methods both in the construction as well as evaluation of the designed artifact. This can for example be in the form of mathematical formalism when describing the created artifact. In the construction step the rigour is about assessing the artifact in respect to the applicability and generalizability of the artifact. Though, rigour should not be emphasised too much as it can reduce the relevance of the research, but it is necessary for IS research to be both rigorous and relevant.

Rigour is derived from the effective use of the knowledge base, by using the theoretical foundations and research methodologies. In evaluating the artifact rigour is about making sure the evaluated performance metrics is appropriate. (Hevner et al., 2004.)

The sixth guideline, design as a research process, says that design science research is inherently iterative. It is not possible to make the perfect design in one go. Instead the design should be iteratively improved, it's a search process to find an effective solution. Simplifications of the problem can be used to first create a solution that is then improved. The problem can also be split into smaller problems that are solved one at a time. In design science, due to the complex nature of the problems, it is not possible to envision all the possible solutions, rather finding a solution needs to be approached as a process building towards a solution. (Hevner et al., 2004.)

The seventh and final guideline, the communication of research, is a guideline for how to report the done research. Results in design science need to be presented both to technology-oriented and management-oriented people. The technical people need enough details to implement the presented artifact in their organizational context, if the research is valid for their context. This allows practitioners to benefit from the research and other researchers to build on top of the existing research. Management people need enough details to determine whether their organization should use resources to construct or purchase the artifact. For this audience design science research should focus on reporting on the required knowledge to effectively apply the artifact, more than describing the details of the artifact. One way to achieve this communication to managerial audiences is to present these details in a well-organized appendix. (Hevner et al., 2004.)

Besides the guidelines, the other important part of design science research is the research framework, that provides context for the research. An overview of this framework is shown in Figure 1. Hevner et al. (2004) say that the framework they present is important, because it represents a feedback loop that allows new ideas to flow back into design science research. This is, according to them, very important in enabling further innovations in the design science field, because information system research is at the intersection of people, organizations, and technology. Behavioral-science research produces theories that seek to predict or explain what happens when artifacts are used. These predictions have mainly been done on instantiation artifacts, concrete systems, but the methods could also be applied to other artifacts as well, for example model artifacts. (Hevner et al., 2004.)

The first feedback loop in Figure 1 goes from the environment to the research and back. The environment defines the problem space, and in information system research it consists of people, organizations, and their existing and planned technologies (Hevner et al., 2004). The information coming in from the environment to the research are the business needs, what is needed in the industry. They are used to explain the relevance of the research (Hevner et al., 2004.), so that the research is not exploring some totally random topic that has no real world relevance. This is included in the framework in order to ensure that the research has utility (Hevner et al., 2004). The design science research contributes back to the environment when the results of the research are applied in the environment. Once implemented, the experiences from that feed back into the knowledge base as well as practice (Hevner et al., 2004).

The second feedback loop goes from the knowledge base, the existing literature, to the research and then back. The previous work is used to take advantage of in future research, for example by using existing methodologies or building on top of existing artifacts (Hevner

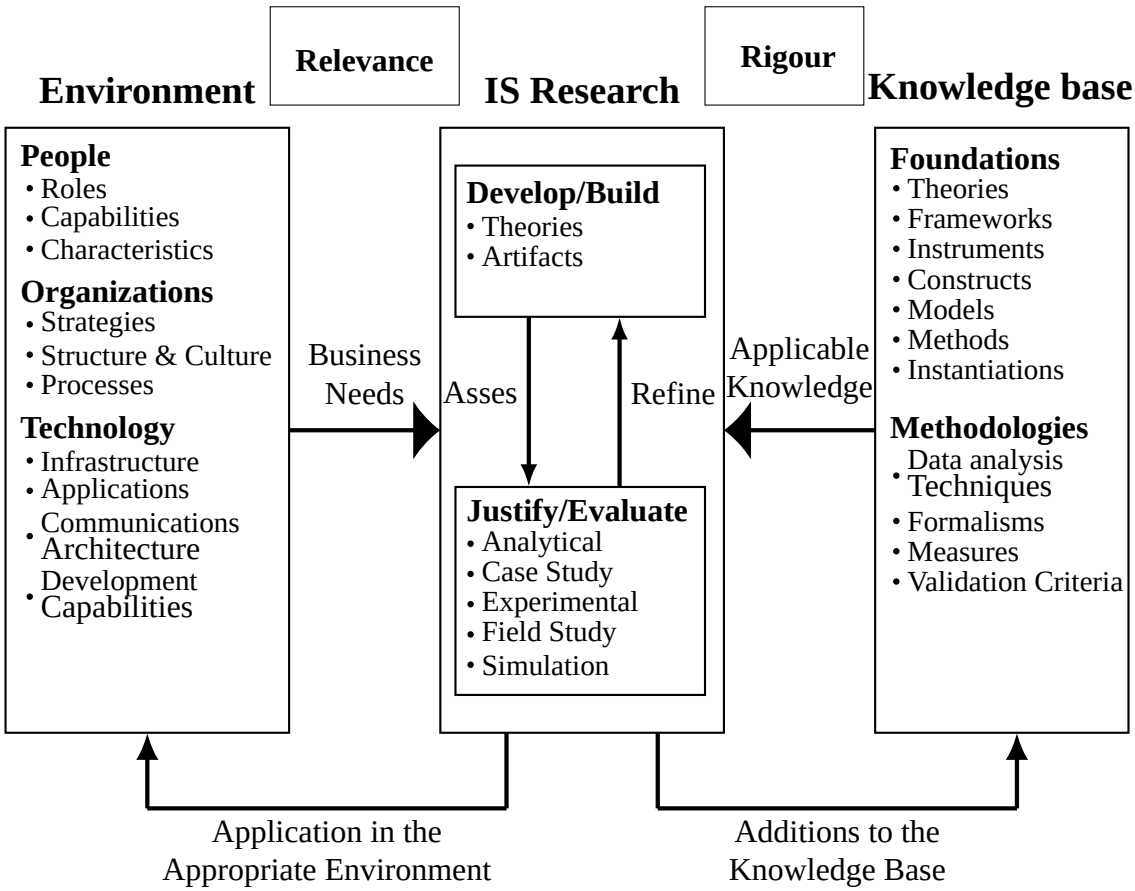

Figure 1. Design science research framework as presented by Hevner, March, Park, and Ram (2004), with slight formatting changes to fit.

et al., 2004). When the existing theories and methodologies are taken into account, it brings rigour into the new research. The completed design science research then contributes back to the knowledge base when the results are communicated back, for example in the form of an article. In design science research the quality of artifacts is most often verified with computational or mathematical methods, but empirical techniques are also sometimes employed. (Hevner et al., 2004.)

Inside the research process there is also a loop, where successive versions of the artifacts or theories are created, and then they are evaluated. This is so that weaknesses in the artifacts can be identified and improved on (Hevner et al., 2004). Then based on the evaluation a new version of the artifact is created. So this is very similar to iteratively building software.

In the early stages of a discipline or with significant differences in the environment, creating new artifacts relies much more on creativity as well as trial-and-error than existing knowledge. Once design science research has resulted in best practices being contributed to the knowledge base, the system building becomes a routine application of the knowledge, to known problems. This is why straight up implementing a new system is not considered research. Whereas design science research addresses important unsolved problems, with new solutions, or improves existing solutions in some significant way. (Hevner et al., 2004.).

The framework presented by Hevner et al. (2004), as shown in Figure 1, was modified to

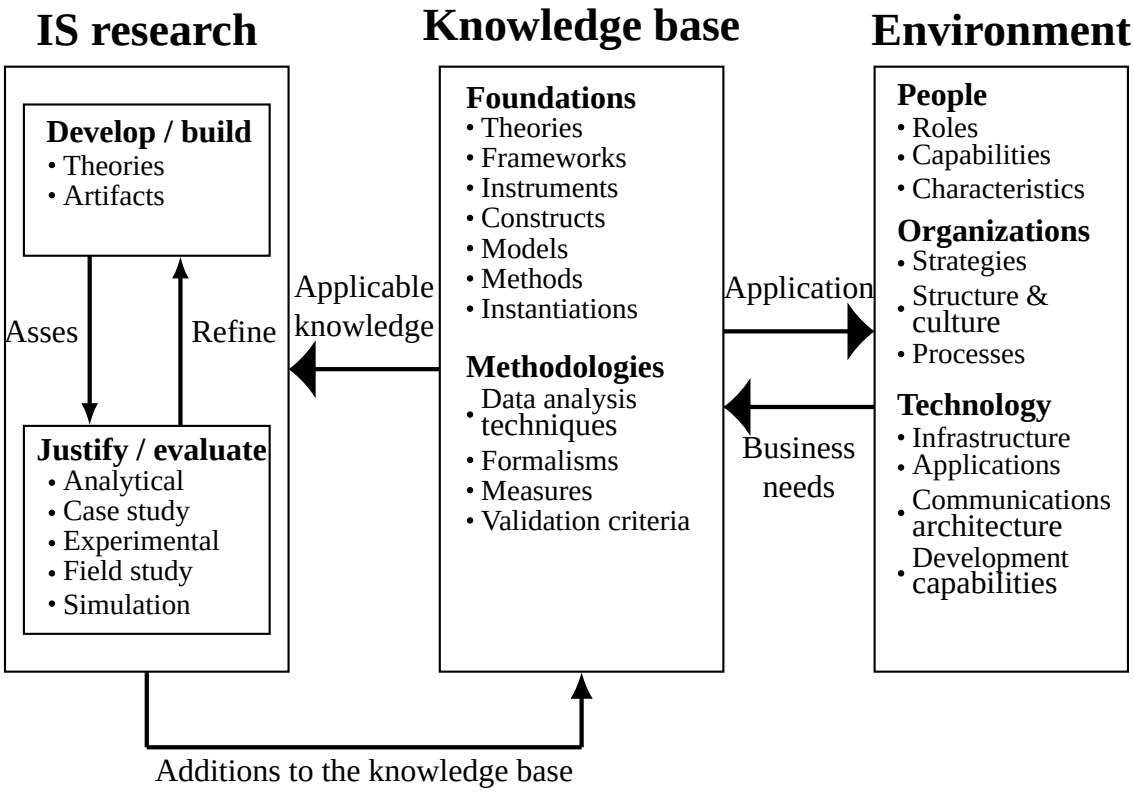

Figure 2. Modified design science research framework adapted from Hevner, March, Park, and Ram (2004)

fit this research better. The modified framework overview is shown in Figure 2. The main difference is that the environment part is changed. In the model by Hevner et al. (2004) the environment contains the context and provides the problems for study, that are related to people, organizations, and technology, forming a feedback loop with the research. In the modified model, this research did not directly interact with the environment. Instead this research only previous research was taken into account. No businesses were cooperated with to gather data from the field. Instead problems identified in the literature review are used as the justification for the artifact created in this thesis. This thesis contributes to the knowledge base of research by communicating how the new tool was designed, what it can do, and findings from the process of developing the tool. So the other feedback loop with research is followed and just the feedback loop with the environment is altered to be through the knowledge base.

The fist guideline, designing as an artifact, is followed by this thesis presenting an instantiation, the tool, and also a description of the tool in the form of a model. Including these parts will satisfy the requirement of designing as an artifact. The second guideline about relevance is fulfilled by the referenced earlier work used as the basis for this thesis. The third guideline about evaluation is followed in Chapter 6 where the created tool is evaluated using existing static analysis test cases against other static analysis tools.

The fourth guideline is followed in the last two chapters where the research contributions of this thesis are discussed. The fifth guideline was taken into account while developing the tool by keeping applicability and generalizability in mind. Also related to this the performance of the new tool was evaluated with benchmarks in Chapter 6 as well as with the test cases used to develop the tool. The new tool was developed iteratively by expanding the number of successful test cases one by one, while making sure the existing ones did

not break, in order to gradually improve the effectiveness of the tool. This fulfils the sixth guideline. The seventh and final guideline is fulfilled by the existence of this thesis which serves as the method this performed research is communicated.

Additional literature for this thesis, on top of the literature found for Hyyryläinen (2019), were mainly found when searching for information about existing static analysis tools and especially when searching for information related to Clang analyzer. Moerman (2018) was found by searching “c static analysis performance test suite” on Google.

5. New Static Analyser

In this chapter, the design of the new tool is discussed as well as how it was tested. In the next chapter, the performance of the tool is evaluated. The tool developed in this thesis has been named Static Memory Analyser for C++ (SMACPP) and will be referred to by that name in this chapter. Despite the name, the tool is only used on C code because that is what the used test cases consist of.

5.1 Goals

This section outlines the goals that were set for the new tool to achieve.

The primary goal of the new tool is that, it needs to detect memory related issues in a certain context. The context is that it needs to use static analysis on C code and detect problems that Clang's static analyzer cannot detect. The tool also needs to be open source in order to be available for further development by others, unlike many of the previous literature that do not give their source code away. Specifically, buffer overflows are the types of issues to be found, as Clang's analyzer cannot detect even very simple cases, which turned out to be easy to handle when developing the new tool.

Currently, Clang's own analyser cannot detect even simple buffer overflow problems, but this should be improved by making the new tool be able to detect issues that Clang cannot. So the requirement is to make a complementary tool to Clang so that the combined use of both tools can detect additional issues, but should not increase false positives as that decreases the utility. One more requirement for the new tool is that it should be easy to incorporate in existing workflows so that it is useful in practice.

What was not considered a requirement, was making sure the tool is forwards and backwards compatible with different Clang versions. Optimally a production ready tool would be made so that it can work with multiple different Clang versions, but currently the new tool was made to work with just the one Clang version it was developed against. In fact it turned out that when updating Clang to a newer major version some code changes were needed to make the tool compile again. Clang documentation says that the Clang tool API is made to be more stable, however that does not allow the same kind of integration and features as the plugin approach, so it was not used.

5.2 Background

This section provides background information on the concepts and technologies that are useful in understanding the next section, the design of the new tool.

Clang is a compiler for C and C++. It is open source and it even provides some facilities for developing software that uses Clang functionality like a library. It can for example help

with parsing source code correctly in order to make it easier to implement tools that need to understand a language that Clang can compile. Clang is based on the LLVM project, which provides the actual machine code generation and low level optimization. So in terms of compiler terminology this makes Clang a frontend for LLVM adding support for additional input languages to it. In this thesis Clang is used to help in parsing the source code as there would be much more work to make a static analysis tool if you also need to write the source code parser as well, instead of using an existing parser.

The parsed form available from Clang is called an abstract syntax tree (AST), it is an abstraction of the source code. See Appendix B for an example of the kind of AST that Clang creates. In a compiler, the parsing phase converts a stream of tokens, derived from the source code, into a parse tree (Safonov, 2010, Chapter 4). The AST is a tree structure where parse tree constructs are replaced with higher level constructs omitting unnecessary lower level details, and capturing semantics to make later operations in the compiler easier (Safonov, 2010, Chapter 4). While Clang's AST is a higher level abstraction compared with source code, it is still missing most of the language semantic analysis, meaning that the AST does not fully provide information like which variables are referenced where, this is left up to the new tool for deciphering the semantics of the code represented by the AST. However, the AST does contain some semantic information. It would have been nice to be able to get this semantic analysis from Clang, but that did not seem possible, thus SMACPP uses the AST from Clang, and implements the semantic analysis itself. The Clang AST, for example, does not include variable scope information.

Test driven development (TDD) is a development approach where test cases are written before the code. A cycle in TDD starts by selecting and understanding a requirement and writing a test for it that can fail (Madeyski & Kawalerowicz, 2013). After writing just the test cases, it is checked that the test cases do not pass. Then the actual code to make the test pass is written. Only enough code should be written so that the tests turn from failing to passing. After that new tests need to be written and the cycle repeated. This means that no code should be written before there is a failing test. The effect is that the tests drive the development process (Madeyski & Kawalerowicz, 2013). When developing this new tool, the development was driven forward by implementing code to pass one new test case at a time. It could be argued that the test cases were too big, because a lot of code needed to be written to pass each additional test, and so it can be argued that TDD methodology was not followed properly. However, the tests did still drive the features that were developed for the tool, thus it can be argued that the spirit of TDD was followed.

5.3 Design

SMACPP is not designed to be a standalone tool for all static analysis needs, instead it is designed to be complementary to Clang's static analyzer. A full integration is not provided in the implementation so any build scripts will need to run the two tools separately in order to get coverage for the different issues that they detect. It would be quite simple to make a wrapper that runs both Clang analyzer and SMACPP with a single command. A slightly more involved solution would be to make SMACPP run Clang static analyzer functionality by directly calling the code implementing that functionality.

SMACPP is implemented as a plugin for Clang. This makes parsing the source code really simple as Clang handles parsing the source code, handling C and C++ code, and building

an abstract syntax tree (AST). The AST is used as the starting point of the new analysis tool. The first part of the analysis is implemented as Clang AST visitors[1]. The visitors build a higher level abstraction of the program only keeping operations important for the analysis phase such as variable assignments, array access, function calls, and conditionals. Then finally this higher level representation is executed in a symbolic fashion checking for unsafe operations.

The overall architecture of SMACPP is shown in Figure 3 in the form of a UML class diagram. Many less significant details have been left out to make the diagram fit on a single page. The main points of the figure are that the program is split into two major parts: the parsing of the abstract syntax tree from Clang, and the actual analysis part that uses an abstraction created by the parsing part. The main class in the analysis part is the Analyzer class which handles the analysis run and uses the other classes on the analysis side. Many of the "uses"-type relationships between the classes have been omitted to make the figure clearer. In the parsing part the most important class is the CodeBlockBuildingVisitor that handles creating the CodeBlock objects by using a lot of recursive visitors that find the needed information from the abstract syntax tree. These visitors handle building the program representation that the analysis part uses.

How all the architecture parts interact with each other is described with the help of an example analysis run in Section 5.5. That section also has a sequence diagram using the parts shown in Figure 3. There is also a textual walkthrough of how an analysis run with SMACPP happens, which will hopefully further make it easier to grasp how the tool works.

At the start of this project making a Clang analyzer plugin was investigated. After a lot of investigation and being stuck on getting started on implementing the analysis tool, it was determined that Clang static analyzer plugin API does not provide the required callbacks in order to allow capturing variable states in order to detect buffer overflows. Granted, it *may* be possible that something was overlooked and that it is in fact possible to create the current SMACPP functionality as a Clang analyzer plugin. This would have many benefits like likely better Clang AST processing and ready-made conditional expression handling. This is because the Clang analyzer API does not work on the AST, instead it has a further level of abstraction on top of it that the analyzer plugins work with. In the long run, this might be the optimal choice, if it turns out that AST parsing and conditional expression handling takes a lot of work, which did take quite a big portion of the total development time.

The development of SMACPP was done iteratively roughly following the principles of test driven development (TDD). The first few iterations were done with tests to get Clang to load SMACPP as a plugin and running it. After that was working, focus shifted on getting the test cases by (Moerman, 2018) working. The first test case to be selected was "overflow/01_simple_if.c", because it was found that Clang's analyzer could not handle any of the cases in the strings overflow category. It also made sense to start from the most basic test and go from there. After the first test was passed by SMACPP the next. These were the next test cases used from the overflow category: "02_simple_if_int1.c", "02_simple_if_int2.c", and "04_simple_switch.c". A lot of code or changes were needed to pass each subsequent, selected test case, so it can be argued that TDD was not fully followed. However the test cases still did direct the development of SMACPP, which is the

[1] See for example Kanjilal (2017) for a briew introduction to the visitor pattern.

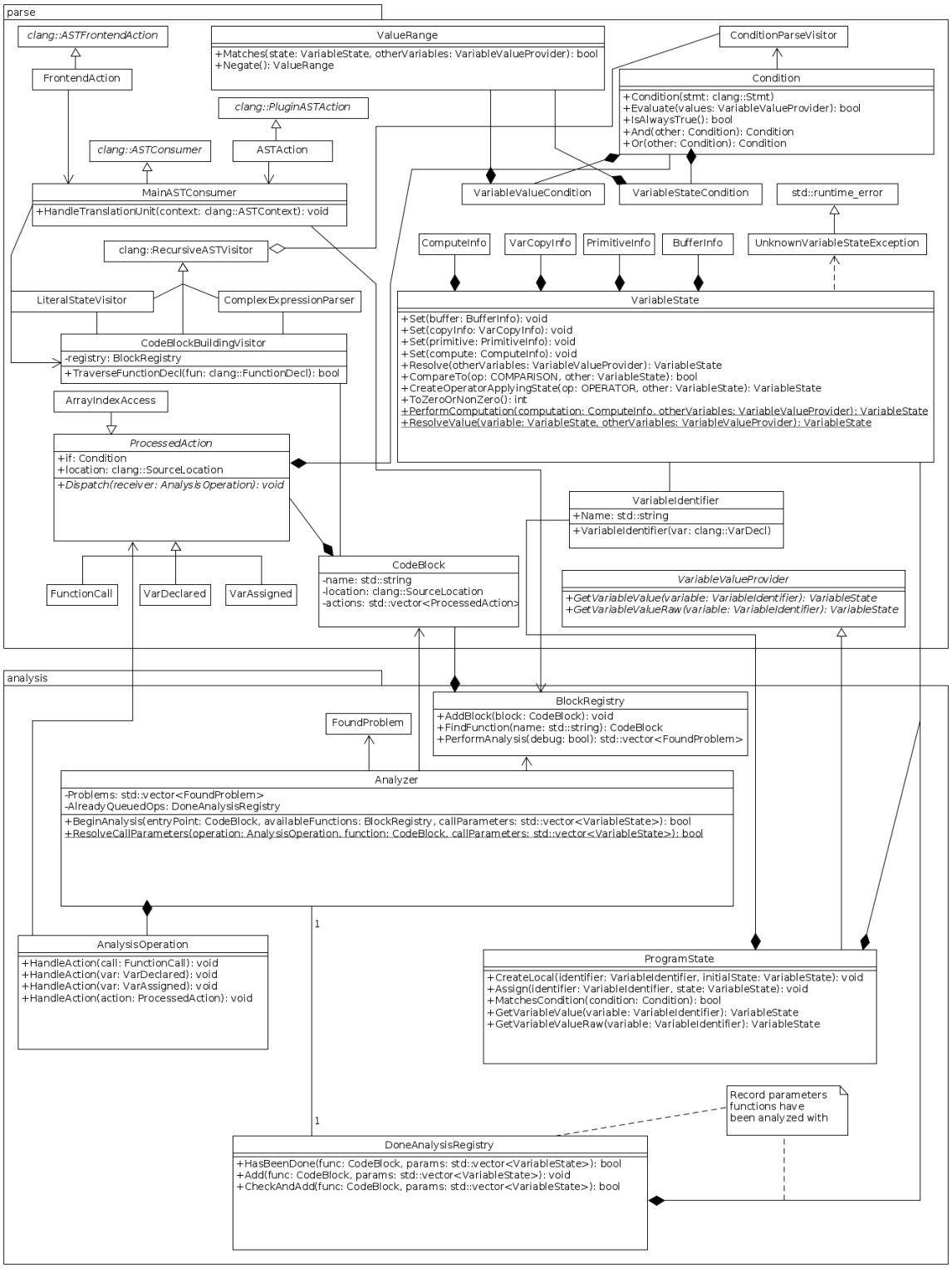

Figure 3. UML class diagram of SMACPP

most important part of TDD. Development was halted after the fourth test due to the next test cases needing large changes to SMACPP and support for new language constructs. One of the big missing features is the ability for functions to return allocated memory, currently SMACPP does not track this.

The analysis phase of SMACPP is composed of running concrete analysis operations, which are functions, that are called with certain parameters. The analysis performs dedu-

plication in order to run each function with the same parameter values only once. This protects against recursive functions taking forever to analyse. The analysis begins by adding “main” function to the list of function calls that need to be analysed. This means that this tool has the same limitation as Frama-C of only working on full programs (Moerman, 2018). It would not take much work to expand the analysis to address this by, for example, starting analysis from all functions that take no parameters, or perhaps even from all functions. But in the latter case the analysis would be much more inaccurate, as the function parameters would all be unknown, and the current analysis is not made to cope with that well. This kind of change would make the analysis work better when analysing code libraries.

The AST visiting phase’s job is to turn the AST into a higher level representation that is used by the actual analysis phase. This representation consists of a few different kinds of actions like variable state changes, array accesses, and function calls. This translation decouples the analysis code from Clang’s AST code. This split was made to make the analyser more resistant to changes in Clang’s source code. It would also be possible to make different front ends for the analyser that would generate the high level abstraction a different way. These high level actions are associated with the function they are contained in. The result of the AST visitor running is an abstract representation of a function from the source code.

Then these functions represented as a list of actions are collected in a simple database. The conditional execution information is embedded in these actions. The action’s condition must be true when the analyser reaches it, in order for the action to be executed. Currently, the database assumes function names to be unique, which does not apply to C++. This is one more aspect where further work is needed to fully support analysing C++, but as said before the test cases only contained C code so this was not a limitation yet. The database could also be expanded to allow saving the functions in it on disk and loading previously saved definitions. This would allow the tool to be extended to support analysing programs consisting of multiple files where function definitions are not visible in all translation units.

The analyser’s symbolic execution works by creating an empty program state for each function to be analysed. The state is first initialized with the function call parameters, then the execution starts. The execution goes through all the actions in the function at each step evaluating the action’s condition and if the condition is true, executing the action. There is potential for performance improvement here by making sure each action is only checked once[2]. Any future static analysis tool writers should take note that this was not a good design decision and instead the abstract actions should have been formed into a directed graph structure. In which, the conditionals would control which edges the analysis follows and which ones it skips. This design would avoid all the known problems with the current design.

The program state in the analysis phase currently consist of just local variable states. This variable state representation is able to represent primitive values, like numbers, memory buffers, and also unknown values. It can also represent variable states that need to be resolved before being used, the first one of these is a state representing variables being assigned to each other, in order to allow the analysis to have better branch handling. The second form is allowing arithmetic operations on two variable states in order to handle

[2]There is quite likely a bug in the code here if some code inside a conditional statement modifies a variable part of the condition, the rest of the conditional actions that should have been executed are skipped. The looked at test cases did not contain this kind of code so this got ignored.

some basic current variable state dependent calculations. The representation does not represent the values stored in memory buffers, neither does it represent more complex values like structs or classes. This leaves potential for conditional expressions that the analyser cannot understand. There are test cases where modeling the contents of memory buffers would be needed to pass them.

Also missing from the variable state analysis is aliasing. This means that SMACPP does not do alias-sensitive analysis. The result of this is that if there are two pointers that refer to the same memory location, and the value is modified through one, but a conditional branch reads the value through the other one (Moerman, 2018, pp. 6–7), SMACPP cannot correctly infer which conditional branch is taken when the program is run. This will lead to worse analysis when a program is analysed which contains this kind of code. None of the in-depth looked at test cases contained this type of code.

Much of the design was directed by the test cases that were chosen to be the ones that SMACPP should be able to handle. Other considerations were largely ignored in the current implementation due to limited resources, so that effort could be focused on getting results in the form of concrete test cases where SMACPP performs better than Clang's analyser, by finding problems in code that Clang cannot correctly handle. Most of these cases were ones where a buffer overflow happened. While the cases were quite simple Clang could not handle them, meaning that Clang analyzer likely has no representation of how big buffers variables reference. This was the main point to implement in SMACPP and add checks for seeing if buffers are accessed with too large indices.

5.4 Implementation and testing

The new tool (SMACPP) was implemented in C++ and Clang APIs were leveraged where possible. This was done to reduce the amount of programming needed. However the usage of the Clang C++ API may need changes to SMACPP when new versions of Clang are released. So far SMACPP has been compiled against Clang 7.0.0 and 8.0.0 with only a couple minor changes needed, so while the API is not guaranteed to be stable, in practice it seems quite stable. This is good news for the longevity of the tool that it can be updated quite easily for new Clang versions. The actual tool part is contained in a shared library that Clang can be instructed to load as a plugin when compiling source code. There is a helper wrapper that automatically adds the plugin loading flags and forwards are the rest of the parameters to Clang. This wrapper can be used, for example, by using environment variables to instruct build tools to use the wrapper as the compiler.

SMACPP was developed iteratively, almost in a Test Driven Development way, implementing the needed functionality to pass one of Moerman's (2018) tests at a time. These tests are available on Github[3]. Focus was given to the tests that Clang's static analyzer could not pass. Looking at Clang's generated AST for each of the tests was very helpful in designing how the AST visiting should work and for determining what kind of information is available to it. These tests guided the development towards making as many of the tests pass as possible, while avoiding writing too overspecific detections that would not be useful in a slightly different version of the test case that might appear in real world code.

[3] https://github.com/JMoerman/JM2018TS

The implementation of SMACPP along with test scripts and a script for downloading the test data is on Github[4]. The repository contains instructions for compiling. After compiling, the test cases can be run with the “Benchmark.rb” script. The code is licensed under the MIT license in order to give maximum freedom to anyone who wants to build on top of the current code.

5.5 Example analysis run

This section uses an example analysis run with SMACPP to explain how the analysis works. The code used for the example run is from the test case set used to develop SMACPP.

The following source code was used in the example run:

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include "string_overflow.h"

void string_overflow_if_else(bool a, bool b) {
    char long_string[] = "this is a long string";
    char short_string[] = "short string";
    char* to_print;


    if(a) {
        to_print = long_string;
    } else {
        to_print = short_string;
    }

    if(b) {
        to_print[11] = '?';
    } else {
        to_print[13] = '?';
    }

    printf("%s\n", to_print);
}

int main() {
    string_overflow_if_else(true, true); /* OK */
    string_overflow_if_else(false, false); /* DANGER */
    string_overflow_if_else(false, true); /* OK */

    return 1;
}
```

[4] https://github.com/hhyyrylainen/smacpp

The source code is copyright (c) 2018 Jonathan Moerman[5]. The AST generated by Clang from that code is included in Appendix B.

The analysis was run with the following command:

```
smacpp -Xclang -plugin-arg-smacpp -Xclang -smacpp-debug -o /dev/
    null -I test/data/JM2018TS/strings/overflow test/data/
    JM2018TS/strings/overflow/test_incorrect/01_simple_if.c
```

This is not much different from a normal SMACPP run, other than the fact that the debug flag is passed to SMACPP and the compiler output is not written to a file, instead it is discarded. This means that SMACPP can be run on a program while it is also built normally at the same time, if the compiler output was not discarded. The SMACPP executable here manages running Clang with the additional parameters needed to load the SMACPP Clang plugin file.

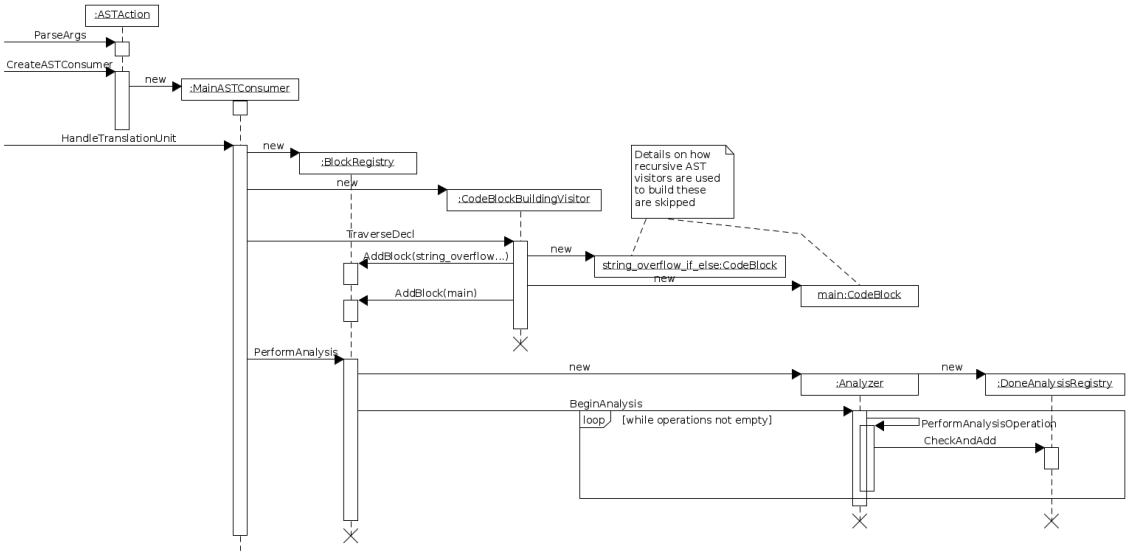

Figure 4. UML sequence diagram showing the main flow of a SMACPP analysis run

The overall flow of execution is shown in Figure 4 in the form of a sequence diagram. Some details have been omitted to keep the diagram easier to read. The run begins by Clang doing its thing and then at some point calling ASTAction in SMACPP to parse the extra arguments for the plugin. After that Clang calls the method CreateASTConsumer, which creates MainASTConsumer and passes that back to Clang. Now when Clang finishes parsing a translation unit, the SMACPP AST consumer is given the AST of the translation unit, then the parsing phase of SMACPP starts.

What happens in the parsing phase is that MainASTConsumer uses quite a few recursive AST visitors, the details of which is omitted from the diagram to keep it readable, to build CodeBlocks out of all the functions in the translation unit. The finished CodeBlocks are stored in a BlockRegistry class instance. After all the CodeBlocks are created the parsing phase is complete. Now the CodeBlocks contain an abstraction of what the program does, in the form of derived classes of ProcessedAction. At this point in debug mode SMACPP prints out the contents of the CodeBlocks as follows:

[5]The code is licensed under the MIT license, see https://github.com/JMoerman/JM2018TS/blob/master/LICENSE for the license details, they are omitted here to conserve space

```
completed block: CodeBlock(string_overflow_if_else):
 params: a b
actions:
 tautology VarDeclared long_string value: buffer of size 21
 tautology VarDeclared short_string value: buffer of size 12
 tautology VarDeclared to_print value: unknown
 a != 0 VarAssigned to_print = assign from long_string
 a == 0 VarAssigned to_print = assign from short_string
 b != 0 ArrayIndexAccess to_print[11]
 b == 0 ArrayIndexAccess to_print[13]
 tautology FunctionCall printf(unknown, unknown)
block end

completed block: CodeBlock(main):
 params:
actions:
 tautology FunctionCall string_overflow_if_else(1, 1)
 tautology FunctionCall string_overflow_if_else(0, 0)
 tautology FunctionCall string_overflow_if_else(0, 1)
block end
```

As can be seen there is one CodeBlock for each of the functions defined in the C code. The actions within the blocks first have a condition that must pass for them to be evaluated. In the end this turned out to be a design mistake, which should be avoided in the future when making static analysis tools. Trees of actions, with conditions determining which branches are executed, seem a much more viable way to implement this kind of analysis. To the right of the condition is the type of the action as well as the parameters of the action. From the CodeBlock for the function “string_overflow_if_else” it can be seen that depending on the values of a and b different variables are assigned as well as different array indices are accessed.

After the parsing phase, the analysis phase starts by the MainASTConsumer calling the PerformAnalysis method on an instance of the Analyzer. Now the analyser builds an AnalysisOperation from the entrypoint CodeBlock and puts it in a queue. Then the analyser calls PerformAnalysisOperation on itself with the first queue item until the queue is empty. The AnalysisOperations can queue more operations if there is a function call in them that has not been checked before. This is done with the call to DoneAnalysisRegistry that checks if the function with the given parameters have been analysed yet or not. The following is the output of the analyser printing each step that it executed:

```
1 analysis at step: tautology FunctionCall
     string_overflow_if_else(1, 1)
2 analysis at step: tautology FunctionCall
     string_overflow_if_else(0, 0)
3 analysis at step: tautology FunctionCall
     string_overflow_if_else(0, 1)
4 analysis at step: tautology VarDeclared long_string value:
     buffer of size 21
5 analysis at step: tautology VarDeclared short_string value:
     buffer of size 12
6 analysis at step: tautology VarDeclared to_print value:
     unknown
7 analysis at step: a != 0 VarAssigned to_print = assign from
```

```
      long_string
 8  analysis at step: b != 0 ArrayIndexAccess to_print[11]
 9  analysis at step: tautology FunctionCall printf(unknown,
      unknown)
10  analysis at step: tautology VarDeclared long_string value:
      buffer of size 21
11  analysis at step: tautology VarDeclared short_string value:
      buffer of size 12
12  analysis at step: tautology VarDeclared to_print value:
      unknown
13  analysis at step: a == 0 VarAssigned to_print = assign from
      short_string
14  analysis at step: b == 0 ArrayIndexAccess to_print[13]
15  analysis at step: tautology FunctionCall printf(unknown,
      unknown)
16  analysis at step: tautology VarDeclared long_string value:
      buffer of size 21
17  analysis at step: tautology VarDeclared short_string value:
      buffer of size 12
18  analysis at step: tautology VarDeclared to_print value:
      unknown
19  analysis at step: a == 0 VarAssigned to_print = assign from
      short_string
20  analysis at step: b != 0 ArrayIndexAccess to_print[11]
21  analysis at step: tautology FunctionCall printf(unknown,
      unknown)
```

The analysis steps have been numbered here to make it easier to reference them. The analysis starts from the main function. Here analysis steps 1-3 are the actions contained in the CodeBlock for main. These operations queue 3 additional analysis operations that are ran starting from step 4. Each of the three function calls begins with 2 steps, 4 and 5, for declaring the short and long string variables, as well as the variable that will point to either one of the strings, which is first declared in step 4. Then there are two steps, 7 and 8, which first copy a pointer, either to the short or long string to a variable, and then an index is accessed through that variable. Under certain conditions, this leads to a buffer overflow. Finally there is step 9, which is a function call to “printf”, which is not very interesting regarding the analysis in this case.

This sequence of steps repeats mostly the same two more times, with the steps that have conditions on them changing between the function calls. The second call begins at step 10 and the third call begins at step 16. In this example step 14 discovers the buffer overflow error present in the analysed code. This is because in step 13 the short variable was assigned to the pointer that step 14 uses.

After the analysis is done, the found problems are printed:

```
test/data/JM2018TS/strings/overflow/test_incorrect/01_simple_if.
   c:21:9: error: Buffer overflow: buffer size: 12 used index:
   13
        to_print[13] = '?';
        ^
1 error generated.
```

The output takes advantage of the error reporting framework built into Clang in order to also point at the rough location the error occurred. Here the location is pointing to the start of the variable name where the buffer overflow happens, but with a little more work the location could be made more accurately to point to the incorrect index number in the source code. In the case that the SMACPP Clang plugin reports errors, Clang will terminate the compilation process, and will not produce an executable as the output. This is similar to errors found by Clang itself. Because of this, it is very useful to hook into the Clang error reporting framework when creating a static analysis tool like this.

6. Evaluation

In this chapter, the performance of SMACPP against Clang's static analyzer and Frama-C is evaluated. The evaluation is done with two test sets Moerman's (2018) and Juliet test suite v1.3 for C/C++[1].

Moerman's (2018) test suite was selected as it was used in their article to compare other open source static analysis tools. It was also very simple to use. Definitely a lot of effort was put into making it as easy as possible to automate running it. The Juliet test suite was selected because it came up in searches as one of the biggest test suites around. It was additionally relatively easy to setup automated tests with. Some corners were cut, though, with the automated runner ignoring the provided list of errors that came with the test suite that also lists the line numbers the problems occur at. Only the buffer overflow related tests, that were not Windows only, were selected from the Juliet test suite as running the whole suite would have taken a lot of computing time time and it would not have given any extra insight to how well SMACPP performs, as it is meant as a buffer overflow detector. The selected Juliet test case sets are: CWE126_Buffer_Overread/s01 and CWE126_Buffer_Overread/s02 In total these include 1836 test cases.

Table 2 contains a summary of the ran test cases with the number of passed tests for each tool, as well as the total time in seconds it took to run the tool on that test case's files. The table contains the results of a single run, based on a few runs the times do not vary a lot between runs so the results of a single run are given here, instead of an average over many runs. More detailed test results for Moerman's (2018) test cases is available in Appendix A it shows the results for the individual tests for all the tested tools.

From Table 2, it can be seen that Clang was the fastest tool to run in all of the cases as well as the one passing most test cases. Second in speed was SMACPP taking about 50% more time than Clang. And the slowest was Frama-C taking 50-100% more time than SMACPP. Frama-C was the tool with the second most passed test cases, with SMACPP only passing the test cases that were explicitly used in its development. It would have been good if SMACPP had passed some tests it was not explicitly designed for, but that did not happen. From a quick look at the SMACPP's failed tests in Moerman's (2018) test cases, it would seem that the tests use different C language features, or features in a more advanced way. This means that it is not surprising that SMACPP did not pass those tests. The tool would need more work done on it to take into account the new features in order to pass more tests.

False positives were detected by the tool emitting an error both for the correct code and incorrect variant. Due to this way of detecting false positives a few false positives from Frama-C are not included in these results. This is because in some tests Frama-C detected an error only in the correct code, meaning that it was counted as a false negative. A few of the false positives of Clang and SMACPP in the Juliet test cases were manually investigated and all of the Clang warnings were unused values and all of the checked

[1] https://samate.nist.gov/SRD/testsuite.php

SMACPP errors were reports of a missing “main” function. It seems that the Juliet test cases are not as uniform as they should be for running automated tests the way these results were obtained. The tests are also unsuitable for static analysis tools that require checking complete programs, ones that start from a “main” function. Frama-C was still the tool with most false positives generated, and all of the manually checked ones were true false positives were Frama-C incorrectly flagged some operation unsafe. Though, not many analysis results were manually checked, as such these conclusions may not hold for all, or even most, of the failed test cases.

Table 2. Summary of test results

	Clang	SMACPP	SMACPP & Clang	Frama-C
Moerman’s (2018) test cases				
Passed tests	55	4	59	24
False positives	0	0	0	10
False negatives	87	138	83	108
Run time (s)	32.1	42.1	77.9	99.2
Juliet test cases				
Passed tests	0	0	0	0
False positives	339	695	695	768
False negatives	1497	1141	1141	1068
Run time (s)	434.2	605.1	927.2	907.1

In Table 2 the column “SMACPP & Clang” has the results for what would happen if Clang’s analyzer and SMACPP were run on the same test case and their reports combined. As can be seen from the first test case results, this works well in increasing the correctly detected issues without reporting any false positives. Also the combined run time is still a little bit lower than Frama-C in the first test cases. In the Juliet test cases, the combined results were not as good due to the previously talked about “false positives” from SMACPP when ran on many of the test cases’ files. In this test case, the combined result was also a tiny bit slower in total run time than Frama-C. However, the run times are very close to each other so there is not a significant difference in the time it takes to run the analysis.

Regarding the combined results, it should be noted that both SMACPP and Clang’s analyzer use Clang to parse the source code. I have not looked through the analyzer source code, but presumably it uses the Clang generated AST, similarly to SMACPP. If this is the case, then this combined approach can be sped up, perhaps even very significantly, to make it more useful, and faster than Frama-C. Further research is needed to determine if this is the case or not. It is possible to run the SMACPP analysis as part of a normal Clang compile, because it is a Clang plugin, which means that if only SMACPP is used for analysis, it can take advantage of the AST tree generated during normal compilation of software saving some time.

When comparing the tools based on which tests they passed, SMACPP performs as planned: it handles cases that Clang fails, and does not pass cases that Clang passes. See Appendix A for these results. From that data it can also be seen that Frama-C passes all of the tests that SMACPP passes. But many, many test cases that Clang passes Frama-C does not pass. However, Frama-C *also* passes many test cases that neither Clang nor SMACPP pass. Frama-C has a high rate of false positives, as discussed previously, which makes it less suitable for being combined with Clang than the combination of SMACPP and Clang. This is because false positives clutter the output of static analysis tools and make it harder

to find real problems.

SMACPP did not pass any of the tests containing loops in the test cases. Only Frama-C were able to pass some of these test cases. As a positive note, none of the tested tools got stuck when trying to analyse a test case with a loop in it. None of the tools got stuck in recursion either. So all tools were able to be run on all of the test files without the tools breaking, but as already covered many test cases were not passed by the tools.

After investigating why none of the tools were able to pass any Juliet tests, it was discovered that the likely reason was that they heavily used “malloc” and related functions. At least that was the most obvious difference. So it seems that none of the tools[2] implement handling of the standard memory allocation functions and track how big memory areas they allocate. SMACPP could be further developed to handle these test cases with two changes: support for keeping track of function return values, and a database of standard functions. That way it could detect that “malloc” creates a new buffer of size that is given as the parameter. Then it would be possible to do bounds checking for it.

[2]This is a known limitation of SMACPP

7. Discussion

In this chapter the findings of this thesis are compared with the prior research and the research question is answered. Theoretical implications are also presented.

7.1 Findings

The goal of this research was to find out how it is possible to create a complementary static analysis tool for Clang's static analyzer that can detect buffer overflows. This was accomplished, as was discussed in Chapter 6. The new developed tool can only detect memory leaks in a few cases as it is still quite simple and does not handle many Clang's AST nodes. This means that on its own it is not very useful, but as an addition to Clang's static analyzer it increases the number of faults that they can detect, without introducing any false positives in the test set. This is good as with many false positives developers may not feel like fixing the problems reported by a tool (Hovemeyer et al., 2005). So the only downside to running it along Clang's analyzer is that it takes extra time to run in that case as both SMACPP and Clang's analyzer will parse the source code, in effect increasing the running time of the combined approach unnecessarily. SMACPP detected some false positives in the Juliet test suite, however the manually checked ones of these were related to missing "main" function, meaning that the actual static analysis logic, once an entrypoint is found, may not have resulted in any false positives. But to confirm that all of the false positives would need to be manually checked, which would require wading through a lot of program output.

While developing the new tool some false starts and design mistakes were made. They are presented here in the hope that they are useful for future static analysis tool developers. The first of these findings is that Clang's static analyzer framework is missing some callbacks for variable operations, meaning that it is not possible to currently write a static analyser plugin that can accurately track what sized buffers variables are pointing to. This is the reason SMACPP could not be implemented as a Clang analyzer plugin, which would have made the implementation easier. The made design mistake, in the design of the new tool, was that each abstracted high level operation was associated with a condition that the program state needed to match for the action to be executed when analysing. This was a mistake as code inside conditionals modifying the conditions now result in incorrect analysis. Additionally the performance impact of constantly evaluating the same conditions might be significant. One positive finding is that Clang's APIs were mostly easy to work with, even with the limited number of tutorials and examples available. One thing that was left unclear is that if it would have been possible to implement the variable evaluation and computation using code from Clang, currently SMACPP implements its own variable value evaluation code as well as condition checking code.

The new developed static analysis tool is pretty basic currently, but it can detect some issues. This is a somewhat similar finding to one of the findings of Hovemeyer et al. (2005), who found that even a simple static analysis tool can be very useful in software

development. This means that it is not necessary to make a very complex tool before it can be useful. Much of the previous literature focused on the importance of using static analysis, for example Sokolov (2007) and Black (2012). Even in this basic form, the new tool could be considered a useful addition to Clang's analyzer, as it did not add any false positives in the first test case set. It could be a useful addition to a code analysis pipeline. Useful static analysis tools need to be able to be incorporated into software development workflows (Ciriello et al., 2013; Dhurjati et al., 2005; Sokolov, 2007). Because of this, a component was included in the new tool, that can be used to run the analysis while compiling a software project, by pointing the project generation tool to use the compiler wrapper component of the new tool. This makes it possible to even fail the compilation of the software due to the analysis finding errors. This is beneficial, as some errors can be caught before even running the software (Das, 2006).

The research question of how can a static analysis tool be created that finds memory usage related issues that Clang's analyzer cannot detect, is answered by this thesis. This text itself serves as the explanation of how such a new tool was designed, built, and tested. Additional insights that will perhaps be useful people building a similar tool are also included. This thesis focuses on a new tool that was built using existing libraries from Clang, so a more general form of the question of how in general to build a static analysis tool, was not covered.

7.2 Implications

The most major implications of this research are that even a simple extra static analysis checks increase the detected true positives of Clang's static analysis without introducing a lot of false positives. The possibility of incorporating the checks from SMACPP into Clang analyzer should be investigated. It likely will not be completely straightforward to include the checks as it was found out that the current SMACPP functionality could not be implemented using the existing Clang static analysis plugin framework. But if these checks could be incorporated, even in an experimental form, it would increase the usefulness of Clang's analyzer. While not exactly a mature tool, the developed tool could have some industry implications. It could perhaps be evaluated or used in an industry setting to find out how effective it is on real world code. People interested in the development of their own static analysis tool, might also find the tool interesting to study or use it as a basis for tools tailored to their needs.

For academia, the implications of this research are that there is a lot of existing work regarding static analysers, but existing open source tools are lacking. That is a big drawback as software practitioners need practical tools they can incorporate in their workflows in order to gain benefits from static analysis, which as explored in existing literature are numerous. Presumably researchers would also benefit from open source implementations that they could use as bases for new work, so that they would not have to rewrite the common static analysis tool parts repeatedly wasting effort. The new tool, SMACPP, developed in this thesis is made open source in the hopes that it can help in future static analysis research efforts.

8. Conclusion

In this chapter, the results and contributions of this thesis are summarised, limitations are discussed and a possible direction for future research is presented.

8.1 What was learned

From the literature review it was learned that, while quite a bit of research has been done on the theoretical basis for static analysis, there is lack of usable tools (Hyyryläinen, 2019). With much of the existing literature not publishing their developed tools, for example (Lee et al., 2018), or the published tools becoming outdated due to lack of updates (Oiwa, 2009). However, the existing open source tools can be useful in detecting some problems, but they have still room for improvement (Moerman, 2018). In this thesis one such improvement was developed showing that it is possible to make such an improvement and it is not extremely difficult to do so. This thesis also responded to the research question *how* it is possible to make such a tool, by sharing the design of the tool as well as some experiences in building it. The source code of the developed tool is available and can be used as a reference on how it works.

8.2 Limitations

A huge limitation of this work is that the developed static analysis tool is still basic and likely not very useful in real world development. More work is needed to be done on the tool before it can be declared as a useful, mature tool for software engineers to use help alleviate all of the problems that were identified by the previous literature, that are caused by not using static analysis. In its current form, the tool has only a little bit of potential in helping in real world software development. Another big limitation is that one of the test case sets used for evaluating the developed tool was also used in the development process of the tool, in a test driven development fashion. This makes the evaluation of the created tool less comprehensive.

One more limitation is the lack of industry partners. This is related to the modified design science methodology framework used in this thesis. The original framework was by Hevner et al. (2004), which included a feedback loop with the industry to better suit their needs. In this thesis, instead of doing a feedback loop with industry partners, the previous found literature was used as the basis for the need that the industry has. Already created static analysis benchmarks were used to evaluate the created artifact in order to further validate that it was not developed in a vacuum, which is a potential issue without the feedback loop.

8.3 Future research

In the future, the tool presented in this thesis, could be further developed in order to make it a fully fledged, industry ready static analysis tool. The experiences presented in this thesis could alternatively be used to help making a new static analysis tool from scratch. In general more design science research should be done on static analysis. Industry partners should also be incorporated in order to create tools that are proven to be useful in software development workflows used in the industry. This work can serve as a basis for such research.

9. Bibliography

Black, P. (2012). Static analyzers: Seat belts for your code. *IEEE Security & Privacy*, *10*(3), 48–52. doi:10.1109/MSP.2012.2

Bodden, E. (2018). Self-adaptive static analysis. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results* (pp. 45–48). ICSE-NIER '18. doi:10.1145/3183399.3183401

Cherem, S., Princehouse, L., & Rugina, R. (2007). Practical memory leak detection using guarded value-flow analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 480–491). PLDI '07. doi:10.1145/1250734.1250789

Chess, B. V. (2002). Improving computer security using extended static checking. In *Proceedings 2002 IEEE Symposium on Security and Privacy* (pp. 160–173). doi:10.1109/SECPRI.2002.1004369

Ciriello, V., Carrozza, G., & Rosati, S. (2013). Practical experience and evaluation of continuous code static analysis with C++Test. In *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation* (pp. 19–22). JAMAICA 2013. doi:10.1145/2489280.2489290

Das, M. (2006). Unleashing the power of static analysis. In K. Yi (Ed.), *Static Analysis* (pp. 1–2). Berlin, Heidelberg: Springer Berlin Heidelberg.

Dhurjati, D., Kowshik, S., Adve, V., & Lattner, C. (2005). Memory safety without garbage collection for embedded applications. *ACM Transactions on Embedded Computing Systems*, *4*(1), 73–111. doi:10.1145/1053271.1053275

Evans, D., & Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE Software*, *19*(1), 42–51. doi:10.1109/52.976940

Frolov, A. M. (2004). A hybrid approach to enhancing the reliability of software. *Programming and Computer Software*, *30*(1), 18–24. doi:10.1023/B:PACS.0000013437.87730.e5

Ganapathy, V., Jha, S., Chandler, D., Melski, D., & Vitek, D. (2003). Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (pp. 345–354). CCS '03. doi:10.1145/948109.948155

Grech, N., Fourtounis, G., Francalanza, A., & Smaragdakis, Y. (2018). Shooting from the heap: Ultra-scalable static analysis with heap snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 198–208). ISSTA 2018. doi:10.1145/3213846.3213860

Heine, D. L., & Lam, M. S. (2003). A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (pp. 168–181). PLDI '03. doi:10.1145/781131.781150

Hevner, A., March, S., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly: Management Information Systems*, *28*(1), 75–105.

Hiser, J. D., Coleman, C. L., Co, M., & Davidson, J. W. (2009). MEDS: The memory error detection system. In F. Massacci, S. T. Redwine, & N. Zannone (Eds.),

Engineering Secure Software and Systems (pp. 164–179). Berlin, Heidelberg: Springer Berlin Heidelberg.

Hovemeyer, D., Spacco, J., & Pugh, W. (2005). Evaluating and tuning a static analysis to find null pointer bugs. *SIGSOFT Software Engineering Notes*, *31*(1), 13–19. doi:10.1145/1108768.1108798

Hyyryläinen, H. (2019, May 22). *Memory error prevention through static analysis and type systems* (Bachelor's Thesis, University of Oulu). Retrieved August 30, 2019, from http://urn.fi/URN:NBN:fi:oulu-201905242071

Iivari, J. (2015). Distinguishing and contrasting two strategies for design science research. *European Journal of Information Systems*, *24*(1), 107–115. Retrieved February 6, 2019, from https://search.proquest.com/docview/1642487097?accountid=13031

Kanjilal, J. (2017, March 17). How to work with the visitor design pattern. *InfoWorld.com*.

Kowshik, S., Dhurjati, D., & Adve, V. (2002). Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems* (pp. 288–297). CASES '02. doi:10.1145/581630.581678

Kroes, T., Koning, K., van der Kouwe, E., Bos, H., & Giuffrida, C. (2018). Delta pointers: Buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference* (22:1–22:14). EuroSys '18. doi:10.1145/3190508.3190553

Landwehr, C. E., Bull, A. R., McDermott, J. P., & Choi, W. S. (1994). A taxonomy of computer program security flaws. *ACM Computing Survey*, *26*(3), 211–254. doi:10.1145/185403.185412

Lee, J., Hong, S., & Oh, H. (2018). MemFix: Static analysis-based repair of memory deallocation errors for C. In L. G. Garci A. Pasareanu C.S. (Ed.), *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 95–106). doi:10.1145/3236024.3236079

Madeyski, L., & Kawalerowicz, M. (2013). Continuous test-driven development: A novel agile software development practice and supporting tool. In *ENASE 2013 - Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering* (pp. 260–267). doi:10.5220/0004587202600267

McGraw, G. (2004). Software security. *IEEE Security & Privacy*, *2*(2), 80–83. doi:10.1109/MSECP.2004.1281254

Moerman, J. (2018, June 24). *Evaluating the performance of open source static analysis tools* (Bachelor's Thesis, Radboud University). Retrieved October 7, 2019, from https://www.cs.ru.nl/bachelors-theses/2018/Jonathan_Moerman___4436555___Evaluating_the_performance_of_open_source_static_analysis_tools.pdf

Oiwa, Y. (2009). Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (pp. 259–269). PLDI '09. doi:10.1145/1542476.1542505

Pomorova, O. V., & Ivanchyshyn, D. O. (2013). Assessment of the source code static analysis effectiveness for security requirements implementation into software developing process. In *2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)* (Vol. 02, pp. 640–645). doi:10.1109/IDAACS.2013.6663003

Safonov, V. (2010). *Trustworthy compilers*. doi:10.1002/9780470593387

Sokolov, S. (2007). Bulletproofing C++ code. *Dr.Dobb's Journal*, *32*(2), 37–42. Retrieved February 1, 2019, from https://search.proquest.com/docview/202693203?accountid=13031

Sun, X., Xu, S., Guo, C., Xu, J., Dong, N., Ji, X., & Zhang, S. (2018). A projection-based approach for memory leak detection. In L. O'Conner (Ed.), *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)* (Vol. 02, pp. 430–435). doi:10.1109/COMPSAC.2018.10271

Weber, M., Shah, V., & Ren, C. (2001). A case study in detecting software security vulnerabilities using constraint optimization. In A. D. Williams (Ed.), *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation* (pp. 1–11). doi:10.1109/SCAM.2001.972661

Wendel, I. K., & Kleir, R. L. (1977). FORTRAN error detection through static analysis. *SIGSOFT Softw. Eng. Notes*, *2*(3), 22–28. doi:10.1145/1012319.1012323

Xie, Y., & Aiken, A. (2005). Context- and path-sensitive memory leak detection. *SIGSOFT Software Engineering Notes*, *30*(5), 115–125. doi:10.1145/1095430.1081728

Xu, Z., & Zhang, J. (2008). Path and context sensitive inter-procedural memory leak detection. In *2008 The Eighth International Conference on Quality Software* (pp. 412–420). doi:10.1109/QSIC.2008.12

Yong, S. H., & Horwitz, S. (2003). Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 307–316). ESEC/FSE-11. doi:10.1145/940071.940113

Appendix A. Test Results

Table 3. Per test case results for Moerman's (2018) test cases. The true and false values are whether the tool passed the test, the failure column contains the reason why SMACPP failed.

Test case	Clang	Frama-C	SMACPP	failure
strings overflow 02_simple_if_int1.c	false	true	true	
strings overflow 01_simple_if.c	false	true	true	
strings overflow 04_simple_switch.c	false	true	true	
strings overflow 02_simple_if_int2.c	false	true	true	
strings unbounded_copy 02_simple_if_int1.c	false	false	false	false negative
strings unbounded_copy 13_loop_for_pointer_arithmetic.c	false	false	false	false negative
strings unbounded_copy 08_loop_for.c	false	false	false	false negative
strings unbounded_copy 14_loop_recursion.c	false	false	false	false negative
strings unbounded_copy 01_simple_if.c	false	false	false	false negative
strings unbounded_copy 04_simple_switch.c	false	false	false	false negative
strings unbounded_copy 18_complex_struct_multiple_methods.c	false	false	false	false negative
strings unbounded_copy 16_misc_pseudo_recursion.c	false	false	false	false negative
strings unbounded_copy 11_loop_while_do_continue.c	false	false	false	false negative
strings unbounded_copy 09_loop_for_complex.c	false	false	false	false negative
strings unbounded_copy 06_simple_pass_by_reference.c	false	false	false	false negative
strings unbounded_copy 15_loop_recursion_multi.c	false	false	false	false negative
strings unbounded_copy 05_simple_goto.c	false	false	false	false negative
strings unbounded_copy 02_simple_if_int2.c	false	false	false	false negative
strings unbounded_copy 12_loop_for_array_branching.c	false	false	false	false negative
strings unbounded_copy 19_complex_refcount.c	false	false	false	false negative
strings unbounded_copy 10_loop_while_continue.c	false	false	false	false negative
strings unbounded_copy 03_simple_if_multi_func.c	false	false	false	false negative

strings unbounded_copy 17_complex_function_pointers.c	false	false	false	false negative
strings unbounded_copy 07_simple_cross_file.c	false	false	false	false negative
strings overflow 13_loop_for_pointer_arithmetic.c	false	false	false	false negative
strings overflow 08_loop_for.c	false	true	false	false negative
strings overflow 14_loop_recursion.c	false	false	false	false negative
strings overflow 18_complex_struct_multiple_methods.c	false	true	false	false negative
strings overflow 16_misc_pseudo_recursion.c	false	true	false	false negative
strings overflow 11_loop_while_do_continue.c	false	true	false	false negative
strings overflow 09_loop_for_complex.c	false	true	false	false negative
strings overflow 06_simple_pass_by_reference.c	false	true	false	false negative
strings overflow 15_loop_recursion_multi.c	false	false	false	false negative
strings overflow 05_simple_goto.c	false	false	false	false negative
strings overflow 12_loop_for_array_branching.c	false	false	false	false negative
strings overflow 19_complex_refcount.c	false	true	false	false negative
strings overflow 10_loop_while_continue.c	false	true	false	false negative
strings overflow 03_simple_if_multi_func.c	false	true	false	false negative
strings overflow 17_complex_function_pointers.c	false	true	false	false negative
strings overflow 07_simple_cross_file.c	false	false	false	false negative
memory double_free 02_simple_if_int1.c	true	false	false	false negative
memory double_free 07_cross_file.c	false	false	false	false negative
memory double_free 13_loop_for_pointer_arithmetic.c	false	false	false	false negative
memory double_free 08_loop_for.c	false	false	false	false negative
memory double_free 14_loop_recursion.c	false	false	false	false negative
memory double_free 01_simple_if.c	true	false	false	false negative
memory double_free 04_simple_switch.c	true	false	false	false negative
memory double_free 15_loop_recursion_multi_alt.c	false	false	false	false negative
memory double_free 18_complex_struct_multiple_methods.c	true	false	false	false negative
memory double_free 16_misc_pseudo_recursion.c	true	false	false	false negative

memory double_free 11_loop_while_do_continue.c	false	false	false	false negative
memory double_free 09_loop_for_complex.c	false	false	false	false negative
memory double_free 06_simple_pass_by_reference.c	true	false	false	false negative
memory double_free 15_loop_recursion_multi.c	false	false	false	false negative
memory double_free 14_loop_recursion_alt.c	false	false	false	false negative
memory double_free 05_simple_goto.c	true	false	false	false negative
memory double_free 02_simple_if_int2.c	true	false	false	false negative
memory double_free 12_loop_for_array_branching.c	false	false	false	false negative
memory double_free 19_complex_refcount.c	true	false	false	false negative
memory double_free 10_loop_while_continue.c	false	false	false	false negative
memory double_free 03_simple_if_multi_func.c	true	false	false	false negative
memory double_free 17_complex_function_pointers.c	true	false	false	false negative
memory access_uninit 02_simple_if_int1.c	true	false	false	false negative
memory access_uninit 13_loop_for_pointer_arithmetic.c	false	false	false	false negative
memory access_uninit 08_loop_for.c	false	false	false	false negative
memory access_uninit 14_loop_recursion.c	false	false	false	false negative
memory access_uninit 01_simple_if.c	true	false	false	false negative
memory access_uninit 04_simple_switch.c	true	false	false	false negative
memory access_uninit 18_complex_struct_multiple_methods.c	true	false	false	false negative
memory access_uninit 16_misc_pseudo_recursion.c	true	false	false	false negative
memory access_uninit 11_loop_while_do_continue.c	false	false	false	false negative
memory access_uninit 09_loop_for_complex.c	false	false	false	false negative
memory access_uninit 06_simple_pass_by_reference.c	true	false	false	false negative
memory access_uninit 15_loop_recursion_multi.c	false	false	false	false negative
memory access_uninit 05_simple_goto.c	true	false	false	false negative
memory access_uninit 02_simple_if_int2.c	true	false	false	false negative

memory access_uninit 12_loop_for_array_branching.c	false	false	false	false negative
memory access_uninit 19_complex_refcount.c	true	false	false	false negative
memory access_uninit 10_loop_while_continue.c	false	false	false	false negative
memory access_uninit 03_simple_if_multi_func.c	true	false	false	false negative
memory access_uninit 17_complex_function_pointers.c	true	false	false	false negative
memory access_uninit 07_simple_cross_file.c	false	false	false	false negative
memory leak 02_simple_if_int1.c	true	false	false	false negative
memory leak 07_cross_file.c	false	false	false	false negative
memory leak 13_loop_for_pointer_arithmetic.c	false	false	false	false negative
memory leak 08_loop_for.c	false	false	false	false negative
memory leak 14_loop_recursion.c	false	false	false	false negative
memory leak 01_simple_if.c	true	false	false	false negative
memory leak 04_simple_switch.c	true	false	false	false negative
memory leak 18_complex_struct_multiple_methods.c	true	false	false	false negative
memory leak 16_misc_pseudo_recursion.c	true	false	false	false negative
memory leak 11_loop_while_do_continue.c	false	false	false	false negative
memory leak 09_loop_for_complex.c	false	false	false	false negative
memory leak 06_simple_pass_by_reference.c	true	false	false	false negative
memory leak 15_loop_recursion_multi.c	false	false	false	false negative
memory leak 05_simple_goto.c	true	false	false	false negative
memory leak 02_simple_if_int2.c	true	false	false	false negative
memory leak 12_loop_for_array_branching.c	false	false	false	false negative
memory leak 19_complex_refcount.c	true	false	false	false negative
memory leak 10_loop_while_continue.c	false	false	false	false negative
memory leak 03_simple_if_multi_func.c	true	false	false	false negative
memory leak 17_complex_function_pointers.c	true	false	false	false negative
memory refer_free 02_simple_if_int1.c	true	false	false	false negative
memory refer_free 07_cross_file.c	false	false	false	false negative
memory refer_free 13_loop_for_pointer_arithmetic.c	false	false	false	false negative
memory refer_free 08_loop_for.c	false	false	false	false negative
memory refer_free 14_loop_recursion.c	false	false	false	false negative
memory refer_free 01_simple_if.c	true	false	false	false negative

memory refer_free 04_simple_switch.c	true	false	false	false negative
memory refer_free 18_complex_struct_multiple_methods.c	true	false	false	false negative
memory refer_free 16_misc_pseudo_recursion.c	true	false	false	false negative
memory refer_free 11_loop_while_do_continue.c	false	false	false	false negative
memory refer_free 09_loop_for_complex.c	false	false	false	false negative
memory refer_free 06_simple_pass_by_reference.c	true	false	false	false negative
memory refer_free 15_loop_recursion_multi.c	false	false	false	false negative
memory refer_free 05_simple_goto.c	true	false	false	false negative
memory refer_free 02_simple_if_int2.c	true	false	false	false negative
memory refer_free 12_loop_for_array_branching.c	false	false	false	false negative
memory refer_free 19_complex_refcount.c	true	false	false	false negative
memory refer_free 10_loop_while_continue.c	false	false	false	false negative
memory refer_free 03_simple_if_multi_func.c	true	false	false	false negative
memory refer_free 17_complex_function_pointers.c	true	false	false	false negative
memory zero_alloc 02_simple_if_int1.c	true	true	false	false negative
memory zero_alloc 07_cross_file.c	false	false	false	false negative
memory zero_alloc 13_loop_for_pointer_arithmetic.c	false	false	false	false negative
memory zero_alloc 08_loop_for.c	false	false	false	false negative
memory zero_alloc 14_loop_recursion.c	false	false	false	false negative
memory zero_alloc 01_simple_if.c	true	true	false	false negative
memory zero_alloc 04_simple_switch.c	true	true	false	false negative
memory zero_alloc 18_complex_struct_multiple_methods.c	true	true	false	false negative
memory zero_alloc 16_misc_pseudo_recursion.c	true	true	false	false negative
memory zero_alloc 11_loop_while_do_continue.c	false	false	false	false negative
memory zero_alloc 09_loop_for_complex.c	false	false	false	false negative
memory zero_alloc 06_simple_pass_by_reference.c	true	true	false	false negative
memory zero_alloc 15_loop_recursion_multi.c	false	false	false	false negative

memory zero_alloc 05_simple_goto.c	true	false	false	false negative
memory zero_alloc 02_simple_if_int2.c	true	true	false	false negative
memory zero_alloc 12_loop_for_array_branching.c	false	false	false	false negative
memory zero_alloc 19_complex_refcount.c	true	true	false	false negative
memory zero_alloc 10_loop_while_continue.c	false	false	false	false negative
memory zero_alloc 03_simple_if_multi_func.c	true	true	false	false negative
memory zero_alloc 17_complex_function_pointers.c	true	true	false	false negative

Appendix B. Example Abstract Syntax Tree

The following is an example abstract syntax tree from Clang when ran on the file “JM2018TS/strings/overflow/test_incorrect/01_simple_if.c” from Moerman (2018). The parts generated from the inclusion of headers have been cut as they are very long. The output has also been manually modified to fit the page by adding line changes and some pointer values were cut to make some of the lines fit.

```
|-FunctionDecl 0x559cd7fff83 prev 0x559cd7ffda2 <test/data/JM2018TS/strings/overflow/test_incorrect/01_simple_if.c:6:1,
|                                                line:25:1> line:6:6 used string_overflow_if_else 'void (bool, bool)'
| |-ParmVarDecl 0x559cd7fff730 </usr/lib64/clang/8.0.0/include/stdbool.h:31:14, test/data/JM2018TS/strings/overflow/
| |                                                test_incorrect/01_simple_if.c:6:35> col:35 used a 'bool'
| |-ParmVarDecl 0x559cd7fff7a0 </usr/lib64/clang/8.0.0/include/stdbool.h:31:14, test/data/JM2018TS/strings/overflow/
| |                                                test_incorrect/01_simple_if.c:6:43> col:43 used b 'bool'
| `-CompoundStmt 0x559cd8000058 <col:46, line:25:1>
|   |-DeclStmt 0x559cd7fff9f8 <line:7:5, col:49>
|   | `-VarDecl 0x559cd7fff928 <col:5, col:26> col:10 used long_string 'char [22]' cinit
|   |   `-StringLiteral 0x559cd7fff9c8 <col:26> 'char [22]' lvalue "this is a long string"
|   |-DeclStmt 0x559cd7fffaf0 <line:8:5, col:41>
|   | `-VarDecl 0x559cd7fffa30 <col:5, col:27> col:10 used short_string 'char [13]' cinit
|   |   `-StringLiteral 0x559cd7fffac8 <col:27> 'char [13]' lvalue "short string"
|   |-DeclStmt 0x559cd7fffb80 <line:9:5, col:19>
|   | `-VarDecl 0x559cd7fffb20 <col:5, col:11> col:11 used to_print 'char *'
|   |-IfStmt 0x559cd7fffcf0 <line:12:5, line:16:5> has_else
|   | |-ImplicitCastExpr 0x559cd7fffbb8 <line:12:8> 'bool' <LValueToRValue>
|   | | `-DeclRefExpr 0x559cd7fffb98 <col:8> 'bool' lvalue ParmVar 0x559cd7fff730 'a' 'bool'
|   | |-CompoundStmt 0x559cd7fffc48 <col:11, line:14:5>
|   | | `-BinaryOperator 0x559cd7fffc28 <line:13:9, col:20> 'char *' '='
|   | |   |-DeclRefExpr 0x559cd7fffbd0 <col:9> 'char *' lvalue Var 0x559cd7fffb20 'to_print' 'char *'
|   | |   `-ImplicitCastExpr 0x559cd7fffc10 <col:20> 'char *' <ArrayToPointerDecay>
|   | |     `-DeclRefExpr 0x559cd7fffbf0 <col:20> 'char [22]' lvalue Var 0x559cd7fff928 'long_string' 'char [22]'
|   | `-CompoundStmt 0x559cd7fffcd8 <line:14:12, line:16:5>
|   |   `-BinaryOperator 0x559cd7fffcb8 <line:15:9, col:20> 'char *' '='
|   |     |-DeclRefExpr 0x559cd7fffc60 <col:9> 'char *' lvalue Var 0x559cd7fffb20 'to_print' 'char *'
|   |     `-ImplicitCastExpr 0x559cd7fffca0 <col:20> 'char *' <ArrayToPointerDecay>
|   |       `-DeclRefExpr 0x559cd7fffc80 <col:20> 'char [13]' lvalue Var 0x559cd7fffa30 'short_string' 'char [13]'
|   |-IfStmt 0x559cd7ffff10 <line:18:5, line:22:5> has_else
|   | |-ImplicitCastExpr 0x559cd7fffd38 <line:18:8> 'bool' <LValueToRValue>
|   | | `-DeclRefExpr 0x559cd7fffd18 <col:8> 'bool' lvalue ParmVar 0x559cd7fff7a0 'b' 'bool'
|   | |-CompoundStmt 0x559cd7fffe18 <col:11, line:20:5>
|   | | `-BinaryOperator 0x559cd7fffdf8 <line:19:9, col:24> 'char' '='
|   | |   |-ArraySubscriptExpr 0x559cd7fffda8 <col:9, col:20> 'char' lvalue
|   | |   | |-ImplicitCastExpr 0x559cd7fffd90 <col:9> 'char *' <LValueToRValue>
|   | |   | | `-DeclRefExpr 0x559cd7fffd50 <col:9> 'char *' lvalue Var 0x559cd7fffb20 'to_print' 'char *'
|   | |   | `-IntegerLiteral 0x559cd7fffd70 <col:18> 'int' 11
|   | |   `-ImplicitCastExpr 0x559cd7fffde0 <col:24> 'char' <IntegralCast>
|   | |     `-CharacterLiteral 0x559cd7fffdc8 <col:24> 'int' 63
|   | `-CompoundStmt 0x559cd7fffef8 <line:20:12, line:22:5>
|   |   `-BinaryOperator 0x559cd7fffed8 <line:21:9, col:24> 'char' '='
|   |     |-ArraySubscriptExpr 0x559cd7fffe88 <col:9, col:20> 'char' lvalue
|   |     | |-ImplicitCastExpr 0x559cd7fffe70 <col:9> 'char *' <LValueToRValue>
|   |     | | `-DeclRefExpr 0x559cd7fffe30 <col:9> 'char *' lvalue Var 0x559cd7fffb20 'to_print' 'char *'
|   |     | `-IntegerLiteral 0x559cd7fffe50 <col:18> 'int' 13
|   |     `-ImplicitCastExpr 0x559cd7fffec0 <col:24> 'char' <IntegralCast>
|   |       `-CharacterLiteral 0x559cd7fffea8 <col:24> 'int' 63
|   `-CallExpr 0x559cd7ffffe0 <line:24:5, col:28> 'int'
|     |-ImplicitCastExpr 0x559cd7ffffc8 <col:5> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
|     | `-DeclRefExpr 0x559cd7ffff38 <col:5> 'int (const char *, ...)' Function 0x559cd7fd3de0 'printf'
|     |                                                                  'int (const char *, ...)'
|     |-ImplicitCastExpr 0x559cd8000028 <col:12> 'const char *' <NoOp>
|     | `-ImplicitCastExpr 0x559cd8000010 <col:12> 'char *' <ArrayToPointerDecay>
|     |   `-StringLiteral 0x559cd7ffff58 <col:12> 'char [4]' lvalue "%s\n"
|     `-ImplicitCastExpr 0x559cd8000040 <col:20> 'char *' <LValueToRValue>
|       `-DeclRefExpr 0x559cd7ffff78 <col:20> 'char *' lvalue Var 0x559cd7fffb20 'to_print' 'char *'
```

That was the AST for the “string_overflow_if_else” function. On the next page the AST for the “main” function is shown.

```
`-FunctionDecl 0x559cd80000f0 <line:27:1, line:33:1> line:27:5 main 'int ()'
  `-CompoundStmt 0x559cd8000490 <col:12, line:33:1>
    |-CallExpr 0x559cd8000230 <line:28:5, col:39> 'void'
    | |-ImplicitCastExpr 0x559cd8000218 <col:5> 'void (*)(bool, bool)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x559cd8000188 <col:5> 'void (bool, bool)' Function 0x559cd7fff830 'string_overflow_if_else'
    | |                                                                                   'void (bool, bool)'
    | |-ImplicitCastExpr 0x559cd8000260 </usr/lib64/clang/8.0.0/include/stdbool.h:32:14> 'bool' <IntegralToBoolean>
    | | `-IntegerLiteral 0x559cd80001a8 <col:14> 'int' 1
    | `-ImplicitCastExpr 0x559cd8000278 <col:14> 'bool' <IntegralToBoolean>
    |   `-IntegerLiteral 0x559cd80001c8 <col:14> 'int' 1
    |-CallExpr 0x559cd8000308 <test/data/JM2018TS/strings/overflow//test_incorrect/01_simple_if.c:29:5, col:41> 'void'
    | |-ImplicitCastExpr 0x559cd80002f0 <col:5> 'void (*)(bool, bool)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x559cd8000290 <col:5> 'void (bool, bool)' Function 0x559cd7fff830 'string_overflow_if_else'
    | |                                                                                   'void (bool, bool)'
    | |-ImplicitCastExpr 0x559cd8000338 </usr/lib64/clang/8.0.0/include/stdbool.h:33:15> 'bool' <IntegralToBoolean>
    | | `-IntegerLiteral 0x559cd80002b0 <col:15> 'int' 0
    | `-ImplicitCastExpr 0x559cd8000370 <col:15> 'bool' <IntegralToBoolean>
    |   `-IntegerLiteral 0x559cd80002d0 <col:15> 'int' 0
    |-CallExpr 0x559cd8000400 <test/data/JM2018TS/strings/overflow//test_incorrect/01_simple_if.c:30:5, col:40> 'void'
    | |-ImplicitCastExpr 0x559cd80003e8 <col:5> 'void (*)(bool, bool)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x559cd8000388 <col:5> 'void (bool, bool)' Function 0x559cd7fff830 'string_overflow_if_else'
    | |                                                                                   'void (bool, bool)'
    | |-ImplicitCastExpr 0x559cd8000430 </usr/lib64/clang/8.0.0/include/stdbool.h:33:15> 'bool' <IntegralToBoolean>
    | | `-IntegerLiteral 0x559cd80003a8 <col:15> 'int' 0
    | `-ImplicitCastExpr 0x559cd8000448 <line:32:14> 'bool' <IntegralToBoolean>
    |   `-IntegerLiteral 0x559cd80003c8 <col:14> 'int' 1
    `-ReturnStmt 0x559cd8000480 <test/data/JM2018TS/strings/overflow//test_incorrect/01_simple_if.c:32:5, col:12>
      `-IntegerLiteral 0x559cd8000460 <col:12> 'int' 1
```